Heidi Kristiina Marsch

Lua-tuen toteuttaminen pelimoottoriin

Opinnäytetyö
Kajaanin ammattikorkeakoulu
Luonnontieteiden ala
Tietojenkäsittely
Syksy 2014

KAJAANIN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖ
TIIVISTELMÄ

Koulutusala Luonnontieteet	Koulutusohjelma Tietojenkäsittely
Tekijä(t) Heidi Marsch	
Työn nimi Lua-tuen toteuttaminen pelimoottoriin	
Vaihtoehtoiset ammattiopinnot	Toimeksiantaja
Aika Syksy 2014	Sivumäärä ja liitteet 45 + 3

Opinnäytetyön tavoitteena oli oppia ymmärtämään, miten korkeatasoisen ohjelmointikielen lisäys tapahtuu C++-kielellä kirjoitettuun pelimoottoriin. Aiheeseen valittiin korkeatasoiseksi ohjelmointikieleksi Lua niminen komentosarjakieli. Opinnäytetyön aihe oli kirjoittajan itselleen antama oppimistehtävä.

Opinnäytetyö käy läpi pelinkehityksessä huomattavat komentosarjakielien ja pelimoottorien vahvuudet ja heikkoudet, jotka ilmenevät niiden yhteisestä ja erillisestä käytöstä. Opinnäytetyön yhteydessä toteutettiin testiprojekti, jolla testattiin opittuja menetelmiä. Testiprojektin tavoitteena oli luoda ja hallita pelimoottorin piirtämiä 3D-objekteja Lua-komentosarjoja käyttäen. Erilliset työn vaiheet olivat 3D-objektin luonti, sen sijainnin muokkaus sekä Lua-komentosarjan kautta rakennetun päivitysfunktion asettaminen ja kutsuminen. Työn toteutuksessa käytettiin Visual C++ Express 2010- ja Visual Studio 2010 -ohjelmia sekä Lua-kielen kehittäjien sivuilta saatavaa Luan omaa virtuaalikonetta. Pelimoottoriksi valittiin Kajaanin ammattikorkeakoulun Pelimoottori II-kurssilla ohjatusti kirjoitettu moottori. Testiprojekti toteutettiin Windows 7 -ympäristössä.

Tuloksena huomattiin, että kirjallisen teorian oppiminen on edelleen suurin ajan viejä uuden asian oppimisessa. Testiprojektissa toteutettiin onnistuneesti sille listatut tavoitteet yhdelle objektille. Tulevaisuuden hyödyn kannalta testiprojektin tekeminen ja sen lopputulos antoivat tarpeellisen perusosaamisen komentosarjakielten ja pelimoottoreiden välisestä kommunikoinnista. Se tarjosi myös tarpeellista tietoa Lua-komentosarjakielen kirjoittamisesta ja hallitsemisesta C++-ympäristössä.

Kieli	Suomi
Asiasanat	C++, Lua, komentosarjakieli, pelimoottori, ohjelmointi
Säilytyspaikka	☒ Verkkokirjasto Theseus ☐ Kajaanin ammattikorkeakoulun kirjasto

KAJAANIN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

THESIS
ABSTRACT

School Natural Sciences	Degree Programme Business Information Technology
Author(s) Heidi Marsch	
Title Implementation of Lua Support for a Game Engine	
Optional Professional Studies	Commissioned by
Date Fall 2014	Total Number of Pages and Appendices 45 + 3

The purpose of this thesis was to learn to understand how a high level programming language is embedded to a game engine, more precisely, to a game engine written in C++ language. The high level language used in the thesis was chosen to be Lua scripting language. The topic of the thesis was chosen by the author as a learning experience.

The thesis goes through the strengths and weaknesses of game engines and scripting languages, whether they are used together or separately. The practical part of the thesis consists of creating a test project to test embedding mechanisms which were learnt during the research. The goal of the test project was to create and control objects drawn by the engine via scripting. The practical part includes the creation of a 3D-object, changing position of the 3D-object, setting a scripted update function for the object, and calling it in the engine. The project was created by using Microsoft Visual C++ Express 2010 and Visual Studio 2010 programs. The virtual machine for Lua was taken from its official website. The project used the author's own game engine which was created during the Kajaani UAS course "Game Engine Project II".

At the end of the thesis it was noticed that the author still has problems in learning new things through reading. During the test project only one object could be created and controlled. For the future benefits, the test project gave necessary know-how on the communication between scripting languages and game engines. It also offered knowledge of Lua-scripting and how to control it in the C++ environment.

Language of Thesis	Finnish
Keywords	C++, Lua, scripting language, scripting, game engine, programming
Deposited at	☒ Electronic library Theseus ☐ Library of Kajaani University of Applied Sciences

SISÄLLYS

SYMBOLILUETTELO

API	Application Programming Interface. suom. Ohjelmointirajapinta.
assosiaatiotaulu	Abstraktinen tietotyyppi. Assosisaatiotaulu on toiselta nimeltä hakurakenne. Se kuvailee avaimia arvoksi.
coroutine	Toiminta, joka sallii ohjelman pysäyttämisen ja jatkamisen.
CPU	Central Processing Unit. suom. suoritin.
CWI	holl. Centrum Wiskunde & Informatica. suom. Matematiikan ja tietotekniikan keskus engl. National Research Institute for Mathematics and Computer Science
DEL	Data-Entry Language.
IDE	Integrated Development Environment. suom. ohjelmointiympäristö.
inkrementaalinen	Tarkoittaa samaa kuin *vähittäin kasvava*.
komentosarjakieli	Yksinkertainen ohjelmointikieli, johon viitataan myös nimityksellä *skriptikieli*. Komentosarjakieli on laaja käsite ja viittaa monenlaiseen skriptaukseen.
konekieli	Tietokoneen prosessorin ymmärtämä kieli, joka koostuu sarjasta konekielisiä käskyjä.
Lua	Komentosarjakieli, jonka nimi tarkoittaa portugaliksi *kuuta*. Koska nimi ei ole kirjainlyhenne, vain sen ensimmäinen kirjain kirjoitetaan suurella alkukirjaimella.
Lua tiimi	Lua-kielen kehittäjäryhmä. Työryhmän jäsenet, jotka ovat suunitelleet ja kehittäneet Luan, ovat Roberto Ierusalimschy, Luiz Henrique de Figueiredo ja Waldemar Celed.
lähdekoodi	Tietokoneohjelman tekstimuotoinen ohjelmointikielinen listaus. engl. source code
MIT-lisenssi	Massachusetts Institute of Techonology lisenssi. Vapaa ohjelmistolisenssi.

mixin	Olio-ohjelmoinnissa oleva luokka, jolle ei voida luoda ilmentymää. Tämä luokka tarjoaa toiminallisuuden periä alaluokasta.
monisäikeistäminen	Rinnakkaisohjemoinnin toteuttaminen käyttämällä säikeitä. engl. multithreading
natiivi koodi	Nimitys konekielelle viitatessa alustariippuvaisiin kielen ominaisuuksien tai kirjastojen osiin.
olio	Olio-ohjelmoinnissa oleva ohjelmiston perusyksikkö.
PUC-Rio	port. *Pontificia Universidade Católica do Rio de Janeiro*. Yksityinen yleishyödyllinen paavillinen katolinen yliopisto, joka kuuluu Pyhän istuimen (paavin) hyväksymiin tai perustamiin yliopistoihin. PUC-Rio on yliopisto Brasilian Rio de Janeiro osavaltion samannimisessä pääkaupungissa.
REPL	engl. read–eval–print. suom. lue-evaluoi-tulosta.
SDK	Software Development Kit. suom. ohjelmistokehityspaketti
skriptaus	Skriptikielen kirjoittamista. Katso kohta *komentosarjakieli*.
SOL	Simple Object Language. Nimi tarkoittaa portugaliksi *aurinkoa*. Vaikuttajana Luan kehittämisessa ja sen nimen määrittelyssä.
syntaksi	Säännöt, jotka määrittelevät oikean yhteenliitetyn symbolien sarjan, jota voidaan käyttää muodostamaan oikein jäsennelty ohjelma käyttäen ohjelmointikieltä.
säie	Käyttöjärjestelmän suorituspolku. engl. thread.
upvalue	Mekanismi Lua-kielessä, joka toteuttaa C-kielen staattisen muuttajan kaltaisen toiminnon tietyssä funktiossa.

1 JOHDANTO

Opinnäytetyön tarkoituksena on toimia todisteena opiskelijan ammattitaidosta ja kyvyistä. Näin ollen päätin valita aiheekseni jotakin, mikä valmistaisi minua mahdollisiin työalan tehtäviin sekä toimisi todisteena kyvystäni hyödyntää jo oppimaani tietoa. Vertailin pelialan yritysten työilmoitusten vaatimuksia sekä Kajaanin ammattikorkeakoulussa oppimiani ohjelmoinnin alueita. Vertailussa nousi esille eräs ohjelmoinnin osa-alue, jonka oman osaamiseni huomasin vaillinaiseksi. Peliyrityksissä arvostetaan korkeatasoisten ohjelmointikielien, komentosarjakielien, osaamista. Valitsin aiheekseni *Lua-tuen toteuttaminen pelimoottoriin*, koska Lua-kielen osaamiselle nousi työilmoituksissa eniten tarvetta yrityksissä. Opinnäytetyön tekemisprosessin tavoitteena on oppia yleisesti komentosarjakielistä sekä tarkemmin Lua-kielen rakenteesta ja toiminnasta.

Opinnäytetyön oppimistavoitteena on selvittää, miten Luan lisäys C++-pelimoottoriin tapahtuu ja mitä siinä on otettava huomioon. Henkilökohtaisina tavoitteinani on oppia Luan toteutuksen periaatteita ja keinoja. Opinnäytetyön tuotoksena eli varsinaisen työn tavoite on toteuttaa Lua olemassa olevaan pelimoottoriin. Työn toteutuksen jälkeen minun on osattava toteuttaa onnistuneesti pelinkehityksen toimintoja Lualla C++-pelimoottorissa. Työn toteutukseen käytetään Visual C++ Express 2010- ja Visual Studio 2010 -ohjelmaa sekä Luan kehittäjien sivuilta saatavaa Luan omaa virtuaalikonetta. Testiprojekti toteutetaan Windows 7 -ympäristössä.

Ennen opinnäytetyön kirjoittamista käytännön kokemukseni Lua-kielestä oli vain sen verran, mitä Kajaanin ammattikorkeakoulun tradenomikoulutuksen ”Luan perusteet” -kurssi on opettanut. Teoriataustani Lua-kielestä on muodostunut siitä, mitä olen oppinut kirjoittaessani seminaarityötä *Luan käyttö peliohjelmoinnissa*. Pelimoottoreihin keskittyneet kurssit Kajaanin ammattikorkeakoulussa ovat valmistaneet minua ymmärtämään pelimoottorin rakennetta ja niiden toimintaa.

Työn teoriaosuudessa hyödynnetään pelimoottoreista ja komentosarjakielistä kertovaa kirjallisuutta. Lisäksi olen lukenut R. Ierusalimschyn kirjan *Lua in programming*, joka käsittelee Lua version 5.2:n sisällön ja käytön. Näillä pohjatiedoilla minulla pitäisi olla tarvittava tietotaito opinnäytetyön aiheen tiedon soveltamiseen. Opinnäytetyö pohjautuu osittain seminaarityöhöni *Luan käyttö peliohjelmoinnissa*.

2 KOMENTOSARJAKIELET

Tässä luvussa käsitellään, mitä on komentosarjakieli, miten se toimii sekä minkälaisia ominaisuuksia se tarjoaa verratessa käännettäviin alemman tason ohjelmointikieliin. Luvun lopussa esitellään joitakin esimerkkejä yleisesti käytetyistä komentosarjakielistä ja kerrotaan niiden yksilöllisistä ominaisuuksista.

2.1 Tarkoitus

Komentosarjakieli on korkean tason ohjelmointikieli. Pelimoottoreiden yhteydessä se tarjoaa käyttäjälle kätevän ja helppokäyttöisen pääsyn pelimoottorin yleisiin toimintoihin. Komentosarjakieli mahdollistaa uuden pelin kehittämisen sekä olemassa olevan pelin mukauttamisen. Sen helppokäyttöisyys mahdollistaa ohjelmoijille ja muille kiinnostuneille mahdollisuuden osallistua pelin kehitykseen. (Gregory 2009, 789.)

Pelin ohjelmointi on ajateltavissa grafiikan ja äänen kaltaisiksi – itse moottorin toiminnasta erillisiksi – medioiksi. Pelimoottoria käsittelevän ohjelmoijan ei pidä joutua selaamaan yksittäisen pelin koodikohtia läpi muuttaakseen moottoria. Samaten peliohjelmoijan ei pidä joutua käsittelemään pelimoottorin syvempiä toteutuksia. Muutettaessa ääni- tai grafiikkatiedostoja pelissä muutosten käyttöön ottaminen ei vaadi koko pelin uudelleen kääntämistä. Kehitettäessä peliä pelin iterointi jättää suurimmalta osin pelimoottorin koodin muuttumattomaksi. Näin ajatellessa huomataan, että kehitettäessä peliä suoraan pelimoottoriin pelin kääntämisessä kuluu paljon turhaa aikaa. Koko pelin kääntäminen pelimekaniikassa tapahtuneen muutoksen vuoksi on epäkäytännöllistä. (Varanese 2003, 9–10.)

Mainitun epäkäytännöllisyyden välttämiseksi sekä toivotun toiminnan saavuttamiseksi pelinkehittäjät ottivat käyttöön komentosarjakielet. Komentosarjan käytön koko konsepti perustuu siihen, että se ajetaan toisen ohjelman sisällä. (Varanese 2003, 19.)

2.2 Ominaisuudet

Komentosarjakieli poikkeaa alemman tason ohjelmointikielistä muutamalla tavalla. Tässä luvussa käsitellään näitä eroja. Koska komentosarjakielet on suunniteltu toteuttamaan yhtäläisiä tavoitteita, ne jakavat keskenään seuraavaksi mainittavien kaltaisia ominaisuuksia. (Ierusalimschy, Figueiredo, Celes 2006, 324.)

Tulkittava kieli

Komentosarjojen kirjoittaminen tapahtuu kuten tavallisesti ohjelmoidessa. Ohjelmoija avaa ohjelmointiympäristön ja kirjoittaa koodin korkeatasoisella ohjelmointikielellä. Ero C++-kielen ohjelmoinnissa ja komentosarjakielen käytössä tapahtuu kielen käännössä. Ohjelmoidessa C++-kielellä ohjelmointikielen kääntäjä muuttaa komennot tietokoneen prosessorille ymmärrettäväksi konekieleksi. Komentosarjakielen kääntäjä ei pysty kuitenkaan tekemään tätä, sillä sitä ei ajeta prosessorin kautta. (Varanese 2003, 19.)

Komentosarjakieli käännetään käyttämällä virtuaalikonetta. Virtuaalikone toimii hyvin samalla tavalla kuin tietokoneen fyysinen prosessori. Virtuaalikoneen kiinnitys tietokoneeseen on kuitenkin virtuaalinen ja se ymmärtää omaa symbolista konekieltään – tavukoodia. Kun koodi esitetään alustariippumattomana tavukoodina, moottori voi helposti kohdella sitä datana. Se voidaan ladata muistiin niin kuin mikä tahansa muukin osa (engl. *asset*) ilman käyttöjärjestelmän apua. Prosessorin tavoin virtuaalikone ottaa vastaan käskyjä suoritettavaksi. Se määrittelee, mitä saatu toimeksianto käskee tehdä ja toteuttaa sen. Virtuaalikoneen suorittaessa koodia, pelimoottorilla on varaa joustaa siitä, miten ja milloin koodi ajetaan. Koska suurin osa komentosarjakielistä on suunniteltu käytettäviksi sulautetuissa järjestelmissä, virtuaalikoneet ovat yleensä yksinkertaisia ja niiden muistijälki on melko pieni. Tulkittavan kielen käyttö mahdollistaa joustavuuden, laiteriippumattomuuden sekä nopean iteroinnin. (Varanese 2003, 19; Gregory 2009, 796.)

Tuki nopealle iteroinnille

Tyypillisen C++-kielellä toteutetun pelinkehityksen työnkulku muodostuu koodin suunnittelusta ja kirjoittamisesta ohjelmointiympäristössä sekä koodin kääntämisestä ja testaamisesta. Tehtävien kiertoa kutsutaan iterointikierrokseksi. Tapahtumasarja palaa aina testaamisesta

takaisin koodin kirjoittamiseen. Iterointikierros voi kestää puolesta minuutista useisiin minuutteihin. Kehitettäessä pelejä konsoleille tai mobiililaitteille aikaa kuluu vielä enemmän, koska tuotos on siirrettävä kohdelaitteeseen testattavaksi. Joka kerta, kun natiivi koodi on muutettava, ohjelma on käännettävä ja linkitettävä uudelleen. Peli on myös sammutettava ja ajettava uudelleen, jotta muutokset tulevat näkyviin. (Petri 2012; Gregory 2009, 796.)

Kehitettäessä peliä komentosarjakielellä ohjelmoijan ei tarvitse huolehtia muusta kuin varsinaisen koodin kirjoittamisesta. Muut aikaa vievät tehtävät ovat poissa. Kun komentosarjakoodia muutetaan, muutosten seuraukset voidaan tavallisesti nähdä hyvin nopeasti. Jotkin pelimoottorit sallivat komentosarjakoodin latautua uudelleen lennosta ilman pelin sammutusta. Toiset moottorit vaativat pelin sammuttamista ja uudelleen ajoa. Siitä huolimatta täysi käännösaika muutoksen teosta ja sen näkemisestä pelissä on tavallisesti paljon nopeampi. (Petri 2012; Gregory 2009, 796.)

Kätevä ja helppokäyttöinen

Komentosarjakielet mukautetaan usein sopimaan tietynlaisen pelin tarpeisiin. Niillä voidaan tuottaa ominaisuuksia, jotka tekevät tavallisista ohjelmointitehtävistä yksinkertaisia, intuitiivisia ja vähemmän virhealttiita. Pelin komentosarjakielellä voi esimerkiksi tuottaa funktioita, jotka mahdollistavat peliolioiden löytämiseen nimeltä. Sillä voi myös toteuttaa muun muassa

- tietyn ajan kulumisen odottamisen
- tilakoneen
- muutettavien parametrien paljastamisen editorissa pelisuunnittelijoille.

(Gregory 2009, 796.)

2.3 Käytön kannattavuus

Pelinkehittäjän kannattaa kysyä itseltään, onko komentosarjakielen käyttö tarpeellista. Komentosarjakieli on erinomainen työväline nopeaan iterointiin, kokeiluun ja myöhemmän muutostyön tekemiseen. Hyödyistään huolimatta komentosarjakielen käytössä menetetään

kuitenkin suoritusnopeutta. Tämä johtuu siitä, että virtuaalikone suorittaa tavallisesti tavukoodinsa ohjeistuksen paljon hitaammin kuin natiivi suoritin suorittaa konekielensä ohjeistuksen. Mikäli pelinkehittäjä ei tarvitse mitään mainituista toiminnoista tai ei ole valmis luopumaan suorituskyvystä, komentosarjakieli ei ole tarpeen. (Rabin 2010, 205; Gregory 2009, 794.)

Valittaessa komentosarjakieltä pelinkehittäjät voivat päättää, ottavatko käyttöönsä kolmannen osapuolen tekemän kielen vai luovatko itse oman kielensä. Monille yrityksille on kätevämpää valita kohtalaisen hyvin tunnettu kieli, jonka käytettävyydestä on hyödynnettävää tietoa. Valmista kieltä käyttävät pelinkehittäjät voivat vielä valita maksullisen ja avoimen lähdekoodin väliltä. Valmista kieltä voidaan käyttää sellaisenaan tai mukailla omien tarpeiden mukaiseksi (mikäli lisenssi sallii sen). Oman kielen kehittäminen tyhjästä ei yleensä ole myöskään sen sekamelskan arvoista, joka sen kehittämisessä syntyy. Kielen kunnossapito tulee kalliiksi, ja uusien työntekijöiden kouluttaminen kielen käyttöön vie myös resursseja. Tämä kuitenkin mahdollistaa kaikkein joustavimman ja mukautettavimman vaihtoehdon. (Gregory 2009, 797.)

Miten valitaan käytettävä kieli

Kielen valitseminen on tärkeää, sillä vääränlaisen kielen käyttö voi tuoda projektille enemmän harmia kuin hyötyä. Pelissä tai moottorissa tarvittavat toiminnot on otettava huomioon valinnassa. Tarvittavien tehtävien listauksen ollessa laaja kannattaa ottaa yleinen kieli. Vaihtoehtoisesti on mahdollista käyttää usean tyyppisiä komentosarjakieliä samassa pelissä. Tällöin joudutaan kuitenkin myös integroimaan ja tukemaan jokaista pelissä käytettyä kieltä. Tavallisesti mitä enemmän kehittäjä on valmis luopumaan suorituskyvystä, sitä enemmän toimintoja ja kehittämisen helpotuksia on mahdollista saada vastineeksi. Kirjoitettaessa suurta määrää pelin koodista komentosarjakielellä virheiden jäljittäjä (engl. debugger) tulee erittäin tarpeelliseksi. (Rabin 2010, 205.)

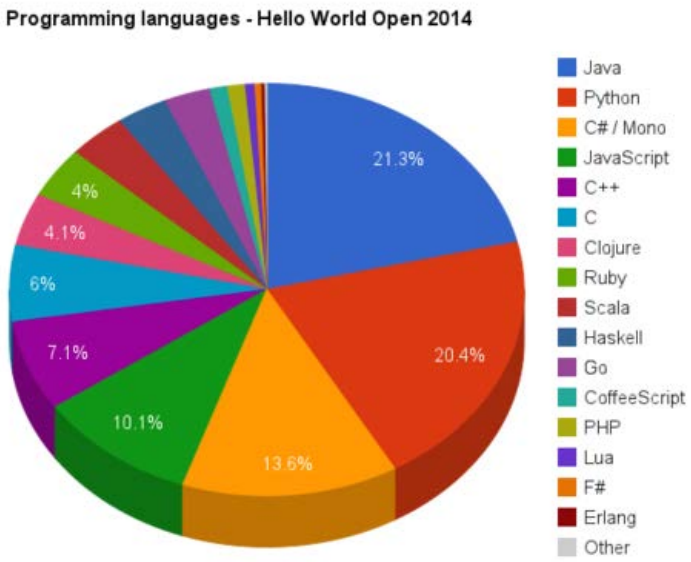
Yksi kielen valintaa rajoittavimmista vaatimuksista on alusta, jolle peli halutaan toteuttaa. Valmiit kielet, joilla on haluttuja ominaisuuksia, eivät aina sovellu tarvittavalle alustalle. Ilman alustalle tarkoitettua tukea kielen käyttäjä voi menettää monia hyödyllisiä ominaisuuksia, esimerkiksi virheidenkäsittelyn. Pelin kehittäjätiimin oma osaaminen on myös ratkaiseva tekijä. Mikäli jollakin pelinkehittäjistä on jo osaamista jostakin kielestä, se tarkoittaa nopeampaa varsinaisen kehittämisen aloittamista. Ohjelmoijien ei silloin tarvitse hidastella uutta kieltä opetellessa. (Rabin 2010, 206.)

2.4 Komentosarjakielityypit

Gregory (2009, 794) jakaa kirjassaan *Game Engine Architecture* pelikomentosarjakielet kahteen yleiseen tyyppiin. Ensimmäinen mainittu tyyppi on nimeltään DLL (engl. data-definition language). DDL:n ensisijainen tarkoitus on sallia käyttäjien luoda ja täyttää tietorakenteita (engl. data structure), joita moottori myöhemmin käyttää. Tämäntyyppiset kielet ovat usein deklaratiivisia. Ne joko suoritetaan tai jäsennetään erillisenä tai ajon aikana, kun data on ladattu muistiin. Toinen Gregoryn kuvaama tyyppi on ajoaikakomentosarjakieli (engl. runtime scripting language). Ajoaikakomentosarjakieli on tarkoitettu suoritettavaksi pelimoottorin kontekstin ajon aikana. Näitä kieliä käytetään tavallisesti laajentamaan tai mukauttamaan kovakoodattua pelimoottorin oliomallia. Sama voidaan toteuttaa myös pelimoottorin muihin systeemeihin. Opinnäytetyössä keskitytään ajoaikakomentosarjakielityypin komentosarjakieliin. (Gregory 2009, 794.)

2.5 Esimerkkejä komentosarjakielistä

Koska kielen valitsemiselle on vaikutus työn lopputulokseen, komentosarjakielten käyttäjät vaativat erinäisiä toimintoja komentosarjakieliltä. Mikäli vaatimukset eivät ole täytettävissä kielen käyttäminen on vähemmän todennäköistä. Tässä osassa tarkastellaan joitakin komentosarjakieliä. Kielistä käsitellään niiden kehittäjää sekä kielestä esiteltyjä ominaisuuksia. Jokainen esimerkkikielistä on kehitetty ja julkaistu 90-luvun alussa. Erilaisten kielten käyttäjälaajuuksien havainnollistamiseksi voidaan katsoa kuvaa 1. Kuva 1 esittää vuoden 2014 *Hello World Open* kilpailun osallistujien kielivalintoja. *Hello World Open* -kilpailu on kansainvälinen ohjelmointikilpailu, jossa tiimit kehittävät tekoälyn kilpa-autolle. Vuoden 2014 kilpailuun osallistui noin 2345 tiimiä. (HELLOWORLDOPEN 2014.)



Kuva 1. Hello World Open 2014 -kilpailun kielivalinnat (HELLOWORLDOPEN 2014)

Python

Python-kieli kehitettiin Amsterdamissa sijaitsevassa matematiikan ja tietotekniikan keskuksessa, CWI:ssa (Scott 2009, 672). Kielen kehitti Guido van Rossum, Alankomaista kotoisin oleva ohjelmointisuunnittelija (Guido van Rossum - Personal Home Page 2014).

Python on proseduraalinen, olioperustainen, dynaamisesti tyypitetty komentosarjakieli. Se on suunniteltu joustavaksi, helppokäyttöiseksi ja yhdistettäväksi muihin ohjelmointikieliin. Python on yleinen valinta pelin komentosarjakieleksi. Taulukko 1 luettelee sille määritellyt parhaat ominaisuudet. (Gregory 2009, 800.)

Taulukko 1. Pythonin ominaisuudet (Gregory 2009, 800–801)

Selvä ja luettava syntaksi	Python-koodia on helppo lukea, koska syntaksi valvoo tietynlaisen sisennetyn tyylin. Se oikeastaan jäsentää tyhjän tilan, jota käytetään tarkoituksella, jotta koodin jokainen rivi voidaan määritellä.
Reflektiivinen kieli (engl. Reflective language)	Pyhoniin sisältyy tehokkaita ajonaikaisia introspektio mahdollisuuksia. Pythonissa luokat ovat ensimmäisen kertaluokan olioita (engl. first-class object). Tämä tarkoittaa, että ne ovat manipuloitavissa ja etsittävissä ajon aikana muiden olioiden tavoin.
Olioperusteinen	Olio-ohjelmointi on rakennettu ydinkieleen. Tämä tekee Pythonin integroinnista pelin oliomalleihin vähän helpommaksi.
Modulaarinen	Python tukee hierarkkisia paketteja. Se kannustaa siistiin systeemisuunnitteluun ja hyvään kapselointiin.
Poikkeuspohjainen virhekäsittely (engl. Exception-based error handling)	Poikkeukset tekevät virhekäsittelykoodin yksinkertaisemmaksi. Se tekee siitä myös lokalisoidumman kuin koodin, joka on tehty poikkeuksia käyttämättömällä kielellä.
Laajat vakiokirjastot ja kolmannen osapuolen moduulit	Python-kirjastoja on olemassa lähes jokaiselle kuviteltavalle tehtävälle.
Sulautettava	Python voidaan helposti sulauttaa pelimoottorin kaltaiseen sovellutukseen.
Laaja dokumentaatio	Python-kielelle on olemassa paljon dokumentaatioita ja oppaita. Näitä on sekä verkossa että kirjamuodossa.

Ruby

Ruby-kielen kehitti japanilainen tietokonetutkija Yukihiro Matsumoto, joka tunnetaan myös nimellä Matz. Kielen ensimmäinen versio kehitettiin vuosina 1993–1995 perustuen Matsomoton osoitukseen sen aikaisten ohjelmointikielten puutteista. Kieli on saanut inspiraatiota useista kielistä, joista yksi on Python. Rubyn uusin versio on 2.1.2, joka julkaistiin 9. toukokuuta vuonna 2014 (Ruby 2014 a). Taulukko 2 kuvaa Rubyn virallisilla sivuilla listattuja Rubyn ominaisuuksia. (Ford 2007, 7.)

Taulukko 2. Rubyn ominaisuudet (Ruby 2014 b)

Kaikki on olioina	Jokaiselle tiedolle ja koodille voi antaa omia ominaisuuksia ja toimintoja.
Joustava	Sallii vapaasti osiensa muuttamisen. Olemassa oleviin osiin voi lisätä itse ominaisuuksia. Tärkeitä Rubyn osia pystyy poistamaan ja muuttamaan.
Lohkot	Tarjoavat paljon joustavuutta.
Mixin	Rubyssa ei ole moniperintää. Se on korvattu *mixin*illä. Luokat voivat saada mixin moduulin toiminnat helposti. Moniperintä on rajoittavampaa.
Muita lisäarvoja tuovia ominaisuuksia	Kuten Python-kielellä, Rubyllä on poikkeuksien käsittely toimintovirheiden käsittelyn helpottamiseksi. Sillä on myös merkitse ja pyyhi (engl. mark and sweep) tyyppinen automaattinen roskienkeräys. C-kielen laajennusten kirjoittaminen on Rubyllä helpompaa kuin Pearlilla tai Pythonilla. Rubyllä on API Rubyn kutsumiseen C-kielestä. Tämä toimii myös Rubyn ollessa sulautettu ohjelmaan komentosarjakielenä.

Java

Java-kielen kehitti Sun Microsystemsille työskentelevä James Gosling. Se kehitettiin, koska laiteriippumattomille sulautettaville kielille oli tarvetta. Kieli kehitettiin vuonna 1991 ja oli julkisesti käytettävissä ensimmäisen kerran vuonna 1995. Oracle Corporation -niminen tietotekniikkayhtiö hankki Sun Microsystemsin (ja sitä myötä Javan) omaan omistukseensa vuonna 2010. Java-kielen ensisijaisia tavoitteita on turvallisuus ja siirrettävyys. Uusin Java-versio on nimeltään Java 1.7 tai SE (standard edition) 7. (Chaudhary 2014, 13, 15, 20–21.) Taulukko 3 listaa Javan ominaisuuksia.

Taulukko 3. Javan ominaisuudet (Chaudhary 2014, 22–25)

Yksinkertainen	Javan syntaksi on melkein samanäköinen kuin C ja C++. Näitä kieliä osaavien ei tarvitse opetella syntaksia tyhjästä. Monet monimutkaiset C/C++-toiminnot on poistettu Java-kielestä.
Turvallinen	Ohjelmat ajetaan Java virtuaalikoneen (JVM) sisällä. Ne ovat muiden osien saavuttamattomissa.
Laiteriippumaton	Java-ohjelmat noudattavat käytäntöä kirjoita-kerran-aja-kaikkialla. Tämä tarkoittaa, että ohjelmat toimivat jopa silloin, kun esimerkiksi käyttöjärjestelmä päivitetään. Windows-alustalla kirjoitettu Java-ohjelma voidaan myös ajaa millä tahansa muulla alustalla, josta löytyy Java-virtuaalikone.
Olioperusteinen	Java on melkein puhtaasti olio-ohjelmoinnin kieli. Suorituskykysyistä se kuitenkin tukee primitiivisiä datatyyppejä (int, double).
Vankkarakenteinen (engl. robust)	Muistin varaaminen ja vapauttaminen tapahtuu Java-kielessä itsessään. Näin poistetaan dynaamisen muistinhallinnan ongelmat, jotka ovat yleisiä C- ja C++-kielissä. Java tukee olioperustaista ajonaikaista poikkeusten käsittelyä. Tämä sallii Java ohjelman palautua ja jatkaa suoritusta poikkeustilan tapahtuessa.
Monisäikeinen	Java-kielellä voi kielen omilla toiminnoilla kirjoittaa monisäikeisiä ohjelmia. Muilla kielillä tämä monisäikeisyys saavutetaan vain hyödyntämällä järjestelmäkutsuja (engl. System call).
Tulkittu ja korkea suorituskyky	Java-ohjelmat ovat tulkittavia, mutta ne ajavat nopeasti verrattuna muihin tulkitsijoihin.
Yleinen	Java on suunniteltu Internetin laajaan ympäristöön. Java-kielellä on sisäänrakennettu tuki moninaisille TCP/IP-pohjaisille protokollille.
Dynaaminen	Jokainen Java-luokka on erillinen suoritus-yksikkö. Luokka ladataan ajon aikana vain, kun sitä tarvitaan.

3 LUA

Tämä luku käsittelee Lua-kieltä, joka on valittu opinnäytetyön toteuttamisen komentosarjakieleksi. Luvussa käsitellään kielen historiaa, ominaisuuksia sekä siitä toteutettuja kolmannen osapuolen tekemiä erilaisia laajennuksia.

3.1 Historia

Luan kehittivät vuonna 1993 Roberto Ierusalimschy, Luiz Henrique de Figueiredo ja Waldemar Celes. Kehitys tapahtui Tecgrafissa, PUC-Rion tietokonegrafiikkateknologian yhtymässä. Tecgraf on laaja tutkimuksen ja kehittämisen laboratorio, jolla on useita teollisuuskumppaneita. Yksi sen suurimmista kumppaneista on brasilialainen öljy-yritys Petrobras. Tämän yrityksen jatkuvat ja vaativat tilaukset saivat aikaan päätöksen Lua-kielen kehittämisestä. Ennen päätöstä kielen kehittämisestä Petrobrasin tilaustöitä toteuttavat tiimit olivat jo tuottaneet omat erilliset ratkaisunsa töilleen. Jotta tilatut sovellutukset voitaisiin valmistaa tehokkaasti, Luiz Henrique de Figueiredon tiimi päätti suunnitella kielen koodatakseen kaiken yhdenmukaisella tavalla. Näin syntyi DEL-kieli, joka salli käyttäjien räätälöidä tiedonsyöttösovellutuksia tarpeiden mukaisesti. Roberto Ierusalimschy ja Waldemar Celes olivat päättäneet käyttää erikoistunutta kuvauskieltä SOL. (Ierusalimschy, Figueiredo, Celes 2007, 3–4.)

Keskusteltuaan molemmista kielistä Luiz Henrique de Figueiredo, Roberto Ierusalimschy ja Waldemar Celes päätyivät seuraavaan lopputulokseen. DEL- ja SOL-kielet päätettiin korvata yhdellä paremmalla kielellä, ja he kolme suunnittelisivat ja toteuttaisivat sen. Luan kehitystiimi oli näin syntynyt. Koska Lua oli korvaamassa DEL- ja SOL-kielet, niistä tuli Luan esivanhemmat sen kehityksen etenemisessä. SOL- kielestä otettiin esimerkkejä syntaksin luonnissa. DEL-kielestä Luiz Henrique de Figueiredon tiimi oli oivaltanut, että suuriakin osia käyttösovellutuksesta pystyttäisiin tekemään sulautettavalla komentokielellä. Tämä oivallus oli ainoa asia DEL-kielestä, joka otettiin huomioon Lua-kielen suunnittelussa. (Ierusalimschy, Figueiredo, Celes 2007, 4–5.)

3.2 Ominaisuudet

Yksinkertaisuus on aina ollut Lua-kielen keskeinen periaate. Kielen nopeus, pieni koko sekä siirrettävyys eri alustoille ovat muodostuneet tästä periaatteesta. Sen yksinkertainen syntaksi syntyi Lua-tiimin huomioidessa Petrobrasin työntekijöitä, joilla ei ollut ammatillista kokemusta ohjelmoinnista. Myös pelinkehityksessä tämä sama yksinkertainen syntaksi on osoittautunut hyödylliseksi. Esimerkiksi suunnittelijat, joilla ei ole kokemusta ohjelmoinnista, pystyvät itse toteuttamaan heille tarpeellisia osioita. (Ierusalimschy, Figueiredo, Celes 2007, 2, 4, 23.)

Luan rakenteellinen yksinkertaisuus perustuu siihen, että sillä on vain yksi tietorakenne käytössään. Tämä tietorakenne on taulu. Luan kielessä nimitystä ”taulu” käytetään assosiaatiotaulusta. Luan taulut tarjoavat toteutuksen moduuleille, prototyyppi- ja luokkapohjaisille olioille, tietueisiin, taulukoille, joukoille, listoille sekä monille muille tietorakenteille. Kaikki toteutetaan siis tauluilla. (Ierusalimschy, Figueiredo, Celes 2007, 2, 24.)

Lua on kirjoitettu C-kielellä niin, että se kääntyy sekä C- että C++-kääntäjillä samalla tavalla. Tähän viitataan myös ”puhtaana C”:nä. Näin ollen Luan käyttöön tarvitaan vain ANSI C-kääntäjä. Kieli kääntyy muuttumattomana useille alustoille (Windows, Linux) ja tarvitsee vain pieniä muutoksia toimiakseen lopuillakin (mobiililaitteet). (Ierusalimschy, Figueiredo, Celes 2007, 2, 12, 24.)

Lualle toteutettu ohjelmointirajapinta sallii täyden kommunikaation Luan ja pelimoottorin kielen välillä. Yhteensopivia kieliä Luan kanssa ovat C- ja C++-kielien lisäksi muun muassa C# (.Net), Java, SmallTalk, Fortran sekä muut komentosarjakielet (esimerkiksi Ruby ja Pearl). Luan helppo lisääminen pelimoottoriin nopeuttaa pelin kehittämisen toteutusta. (Ierusalimschy, Figueiredo, Celes 2007, 2, 24.)

Koska monet pelien alustat vaativat rajatun kokoisia pelejä, on tärkeää, ettei projektiin lisätä mitään ylimääräistä. Komentosarjakielet saattavat helposti tuoda mukanaan ominaisuuksia ja kirjastoja, jotka eivät ole peliprojektin kannalta tarpeellisia. Lua on suunniteltu sulautetuksi sovellutukseen, joten se sisältää vain muutaman kirjaston. Muut komentosarjakielet on tarkoitettu käytettäväksi yksinään, joten ne sisältävät useita kirjastoja. Nopeutensa tähden Lua-kielellä voi kirjoittaa suuriakin koodiosioita. (Ierusalimschy, Figueiredo, Celes 2007 2, 24.)

Kielen poikkeavuudet muista komentosarjakielistä

Verratessa Lua-kieltä edellisessä luvussa esiteltyihin komentosarjakieliin, Luan rakenteessa on joitakin huomattavia eroja. Esimerkiksi Lua ei ole puhdas olioperustainen kieli, mutta se tarjoaa tuen sen tuottamiseen. Keskeinen konsepti Lua-kielen suunnittelussa on tuottaa niin kutsuttuja meta-mekanismeja. Sen sijaan, että ohjelmoijalle tarjottaisiin joukko valmiita toimintoja, Lua-kielessä käytetään toimintojen toteuttamiseen meta-mekanismeja. Luan meta-mekanismit pitävät kielen pienenä ja tuovat säästön käsitteistä. Samalla ne sallivat kielen merkitysopin laajentamisen epätavallisisin tavoin. Vaikka Lua ei ole puhdas olioperustainen kieli, se tarjoaa meta-mekanismeja luokkien ja perinnän toteuttamiseen. (LUA 2014 a.)

3.3 Vahvuudet ja heikkoudet

Tähän osaan on kerätty kommentteja siitä mitä ohjelmoijat ovat kokeneet Lua-kielen hyviksi ja huonoiksi puoliksi. Gregory (2009, 780) kirjoittaa esimerkiksi miten Lua-kielen joustava funktion sidontamekanismi sallii tärkeiden globaalien funktioiden, kuten *sin(),* toteuttaa täysin eri toiminnon. Tällainen tapahtuma ei yleensä ole ollut Gregoryn mukaan toivottua. (Gregory 2009, 780.)

Peliyritys *Almost Human* kirjoittaa pelinsä *Legend of Grimrock* virallisella nettisivulla (2012) toisesta ongelmasta. Luan kanssa on esiintynyt kunnollisten kehittämistyökalujen puute. Lisäksi paikalliset-ja upvalue-arvot dynaamisesti ajetussa koodissa ovat ongelma. Jos uudelleen ladattu funktio viittaa paikalliseen muuttujaan koodin lohkon näkyvyysalueella, uusi funktion versio ei näe niitä. Tämä tapahtuu, koska viitatut muuttujat eivät ole esitelty, kun funktio ladattiin uudelleen. Vain funktion ohjelmakoodi on uudelleen evaluoitu. (Petri 2012.)

Game Development Stack Exchange listaa Lualle joitakin vahvuuksia sekä heikkouksia. *Game Development Stack Exchange* on sivusto itsenäisille ja ammattipelinkehittäjille. Sivustolla voi esittää tiettyjen kriteerien täyttäviä kysymyksiä ja saada muilta sivuston käyttäjiltä vastauksia. Listauksessa olevat vastaukset ovat ohjelmoijien mielipiteitä vuonna 2011 kirjoitettuun kysymykseen: *Mitkä ovat Luan lisäämisen hyvät ja huonot puolet, jos kieli sulautetaan C++-peliin?* Taulukko 4 listaa kysymykseen vastanneiden kirjoittamia Luan vahvuuksia ja heikkouksia. (GAMEDEV 2011.)

Taulukko 4. Luan hyvät ja huonot puolet. (GAMEDEV 2011)

Vahvuudet	
Lyhyt iterointiaika	Dynaaminen ajo on todella nopeaa Lualla.
Konsoli integrointi	Mikä tahansa debug funktio voidaan kytkeä perinteiseen Quake-tyyliseen konsoliin, jolla on REPL-luuppi.
Helppo oppia	Ohjelmointirajapinta on pelkistetty niin, että myös artistit ja suunnittelijat voivat osallistua tehtäviin, jotka usein jäisivät ohjelmoijalle.
Erikoistunut staattinen koodianalyysi	Luan ulostulo on mahdollista analysoida halutessa. Lähdekoodin tiedostot on aika helppo jäsentää, jos käyttäjällä on ohjelmointi käytäntö.
Virhekäsittely	Lualla on hyvät virheenkäsittelyä helpottavat funktiot.
Ohjelmointirajapinnan laajentaminen on helppoa	Uusien funktioiden kirjoittaminen Luan ja C++-kielen väliseen ohjelmointirajapintaan ei ole kovin vaikeaa. On olemassa paljon paketteja, jotka helpottavat tämän automatisoimista.
Yksinkertainen lähdekoodi	Lähdekoodiin on mahdollista tehdä itse muutoksia, jos niin halutaan.
Metataulut	Mahdollistavat monenlaisia asioita tehtäväksi ajonaikana.
Coroutine	Toimintojen pysäyttäminen ja jatkaminen
Heikkoudet	
Arvaamattomasti toimiva automaattinen roskienkeräys	Roskienkeräyksen kertojen ajoittaminen vaihtelee pelistä toiseen.
Tietokoneresurssien käytön kasvu	Suurien datastruktuurien tietorakenteiden tekeminen Luan puolella voi saada aikaan korkean käytön tietokoneresursseissa.
Pirstaloituminen	Pienien muistikoneistojen suurin ongelma.
Tyyppi turvallisuuden puute	Mikäli ohjelmoija ei ole tehnyt hyvää staattista koodin analysointityökalukokoelmaa, voidaan joutua palamaan vanhaan koodiin useita kertoja tekemään ajoaikaista virhetarkastusta.

3.4 Erilaiset Luat

Seuraavat Lua-kielen laajennukset ovat kolmansien osapuolien toteuttamia. Ne eivät ole virallisen Lua-tiimin tekemiä. Näiden laajennusten tarkoitus on helpottaa Lua-kielen käyttöä yksinkertaistamalla käyttäjälle tarpeellisia, mutta vaivalloisia toimenpiteitä.

OOLua

OOLua on C++ sidos (engl. binding) Lualle. Sen tarkoitus on helpottaa Luan sulauttamista C++:aan. Se helpottaa tauluihin käsiksi pääsyä, funktioiden kutsua sekä tyyppien lisäystä ja poistoa pinosta (engl. stack). Se on yhteensopiva Lua 5.1.*, 5.2.*, LuaJIT-1.1.8 ja LuaJIT2 versioiden kanssa. (Code.Google. 2014.)

LuaJIT

LuaJIT on ajonaikainen kääntäjä (engl. Just-In-Time compilation, JIT) Lua ohjelmointikielelle. LuaJIT:iä on onnistuneesti käytetty komentosarjakirjoittamisen väliohjelmistona muun muassa peleissä sekä verkko- ja grafiikkasovellutuksissa. Sitä on käytetty sulautetuissa laitteissa, älypuhelimissa, pöytäkoneissa ja serverifarmeissa. LuaJIT yhdistää joustavuuden korkealla suorituskyvyllä ja erittäin pienellä muistijäljellä. (LuaJit 2014.)

LuaBind

Luabind on kirjasto, joka auttaa luomaan sidonnan C++-kielen ja Luan välille. Sillä on kyky paljastaa C++:lla kirjoitettuja funktioita ja luokkia Lualle. Se myös tarjoaa toiminallisuuden määritellä luokkia Luassa ja antaa niiden juontua toisista Lua- tai C++-luokista. Lua-luokat voivat syrjäyttää virtuaalisia funktioita niiden C++-kantaluokasta. Luabind toimii Luan 5.x-versioissa. (RASTERBAR 2014.)

3.5 Rooli pelialalla

Lua on yksi johtavista pelinkehityksen komentosarjakielistä. Lua-tiimi (2007, 23) toteaa kirjassaan *The Evolution of Lua* tämän johtuvan siitä, että onnistuneimmat kielet on kasvatettu pelkän suunnittelemisen sijaan. Lua-kielen osaaminen on levinnyt pelinkehittäjien keskuudessa kaupaksi käyvänä taitona vuodesta 1998 lähtien. Vuosina 2003 ja 2006 gamdev.net-sivuston tekemät mielipidemittaukset näyttivät Luan kaikkein suosituimpana komentosarjakielenä. Vuonna 2012 *Game Developer* -lehti julisti Luan *Front Line Award 2011* -voittajaksi ohjelmointityökalujen kategoriassa (GDMag Staff 2012). (Ierusalimschy, Figueiredo, Celes 2007 1, 23–24.)

Lua-kieltä on käytetty monissa suosituissa peleissä, mikä on houkutellut yhä useampia ihmisiä tutustumaan Luaan. Suuri määrä pelinkehittäjien antamasta palautteesta on vaikuttanut siihen, miten Lua-kieli on kehittynyt versioidensa aikana. Muun muassa Luan versiossa 5.1 oleva yksinkertainen inkrementaalinen roskien keräys kehitettiin nimenomaan pelinkehittäjien pyynnöstä. Luan siirrettävyys alustalta toiselle on myös yksi syistä Luan laajaan käyttöönottoon. Pelikonsolien kaltaiset rajatut ympäristöt eivät aina tue koko perus C-kirjaston semantiikkaa. Näin ollen Luan käyttämä ”puhdas” ANSI C -toteutus tarjoaa konsolipelien kehittäjille Luan käyttömahdollisuuden pelinkehitykseen. (Ierusalimschy, Figueiredo, Celes 2007 9, 11, 23.)

4 PELIMOOTTORIT

Tässä luvussa käsitellään erilaisia pelimoottorityyppejä. Lisäksi luvussa käsitellään osia, jotka ovat tulleet yleiseksi osaksi pelimoottorin rakenteita. Jotkin kohdassa 4.2 mainitut osat toimivat myös omina moottoreinaan, jotka pelin kehittäjät voivat lisätä osaksi omaa moottorikokonaisuuttaan.

4.1 Tyypit

Pelimoottorin konsepti perustuu tarpeeseen pelkistää (joskus alustariippuvaisesti) yleisten peliin liittyvien tehtävien yksityiskohtia. Pelimoottorit tarjoavat uudelleenkäytettäviä komponentteja, joita voidaan manipuloida pelin elävöittämiseksi. Lataaminen, näytölle tuominen ja mallien animointi, yhteentörmäys, fysiikka, syötteet ja jopa osa tekoälystä voivat olla komponentteina pelimoottorissa. Vastakohtaisesti pelin sisältönä ovat komponentit, jotka tekevät siitä varsinaisen pelin. Näitä voivat esimerkiksi olla yhteentörmäyksen tarkoitus sekä tietynlaiset mallit ja tekstuurit. (Ward 2008.)

Monet pelimoottorit kehitetään ajamaan tiettyä peliä tietyllä alustalla. Jopa yleisempään käyttöön muotoiltu monialustainen moottori on sopiva rakentamaan vain tietynlaisen genren tyyppisiä pelejä. Gregory (2009) toteaa, että mitä yleistarkoituksellisemmaksi pelimoottori tai väliohjelmisto on rakennettu, sitä vähemmän optimaalisempi se on ajamaan juuri tietynlaista peliä tietynlaisella alustalla. (Gregory 2009, 11.)

Viimeisten vuosien aikana talon sisäisten moottoreiden rakentaminen on tullut jatkuvasti kalliimmaksi. Yhä useampi yritys on alkanut erikoistua joko pelikomponenttien tai kokonaisten pelimoottoreiden tekemiseen. Nämä väliohjelmistojen tuottajat voivat myydä tuotteensa peliyritykselle, mikä saa aikaan ”rakenna vastaan osto” -tilanteen. Ward (2008) jakaa artikkelissaan pelimoottorit kolmeen tyyppiin, joilla jokaisella on omat hyötynsä ja haittansa. (Ward 2008.)

Itse kehitetyt pelimoottorit

Hinnastaan huolimatta monet yritykset yrittävät ylläpitää omaa pelimoottoriaan. Tämä tarkoittanee, että ne käyttävät julkisesti olevia sovelluskäyttöliittymiä. Näitä voivat olla XNA:n, DirectX:n ja OpenGL:n kaltaiset ohjelmointirajapinnat. Muita esimerkkejä ovat Windowsin ja Linuxin ohjelmointirajapinnat ja ohjelmiston kehityspakkaukset. Lisäksi kehittäjät voivat käyttää kirjastoja, niin maksullisia kuin ilmaisia, työn helpottamiseksi. Nämä kirjastot voivat sisältää esimerkiksi Havokin kaltaisia fysiikkakirjastoja. (Ward 2008.)

Yleisesti itse kehitetty pelimoottori tarjoaa ohjelmoijalle suurimman joustavuuden. Se antaa valita komponentit, joita halutaan ja antaa integroida ne juuri niin kuin halutaan. Itse kehitetyt moottorit vievät kuitenkin eniten aikaan rakennettaessa. Lisäksi ohjelmoijat joutuvat usein rakentamaan työkaluketjutuksen tyhjästä, sillä pelinkehittäjät voivat harvoin luottaa siihen, että kaikki kirjastot toimivat yhdessä sellaisenaan. Tämä tekee oman moottorin kehittämisestä vähemmän haluttavampaa useimmille pelinkehittäjille. (Ward 2008.)

Lähes valmiit pelimoottorit

Nämä moottorit sisältävät usein muun muassa hahmontamisen, käyttäjäsyötteet, GUI:n ja fysiikat. Esimerkkejä tämäntyyppisistä moottoreista ovat ilmaiset ORGE ja Genesis3D sekä maksullinen Unreal-moottori. Tämän kaltaiset moottorit vaativat vielä jonkin tasoista ohjelmointia, jotta ne saadaan käyttöön pelin tekoon. Ne saattavat vaatia jonkin verran komentosarjan kirjoittamista tai joskus jopa matalamman tason ohjelmointia, jotta varsinainen peli saadaan toimimaan. Lähes valmiit pelimoottorit ovat hiukan rajoittavampia kuin itse kehitetyt pelimoottorit ja ovat usein optimoitu yleistapauksille. Siitä huolimatta monet näistä moottoreista ovat kymmenien ihmisten ja satojen työtuntien tulosta. Vaikka ne eivät tekisikään juuri niin kuin pelinkehittäjä haluaa, ne tarjoavat paremman suorituskyvyn ja vähemmän ponnisteluja kuin itse kehitetyt pelimoottorit. (Ward 2008.)

Osoita ja klikkaa -pelimoottorit

Osoita ja klikkaa -moottorit ovat tulossa yhä suositummiksi. Ne sisältävät täyden työkaluketjutuksen, joka antaa käyttäjän luoda pelin yksinkertaisesti osoittamalla ja valitsemalla haluttuja toimintoja. Tällaisia moottoreita ovat muun muassa GameMaker, Torque Game Builder ja

Unity3D. Ne on rakennettu niin käyttäjäystävällisiksi kuin vain suinkin mahdollista ja vaativat vähimmäismäärän ohjelmointia. Ohjelmoinnin osaaminen ei ole kuitenkaan haitaksi, mutta se ei ole yhtä oleellisesti käytössä kuin aikaisemmin mainituissa pelimoottorityypeissä. (Ward 2008.)

Ongelma osoita ja klikkaa -moottorien kanssa on se, että ne ovat erittäin rajoittavia. Monet moottoreista toteuttavat vain yhden tai kahden pelityylilajin tai grafiikkamallin hyvin. Tämä ei kuitenkaan tee niistä hyödyttömiä. Parasta osoita ja klikkaa -moottoreista on se, että ne antavat käyttäjän työskennellä ja pelata omia pelejä nopeasti, ilman paljoa vaivaa. (Ward 2008.)

4.2 Osat

Tässä kappaleessa käydään läpi erilaisia ominaisuuksia, jotka ovat yleisinä osina pelimoottoreissa. Pelimoottorit koostuvat useasta osasta. Käsitellyistä moottorin osista annetaan aiheen mukainen esimerkki.

API & SDK

API eli ohjelmointirajapinta on yksi kahdesta termistä, jotka esiintyvät usein keskusteltaessa pelimoottoreista. Käyttöjärjestelmät, kirjastot ja palvelut tarjoavat käyttäjälle ohjelmointirajapintoja, jotta niiden tiettyihin toimintoihin voidaan päästä käsiksi. SDK eli ohjelmistokehityspaketti on kokoelma kirjastoja, ohjelmointirajapintoja ja työkaluja, jotka on tehty saatavaksi aikaisemmin mainittujen käyttöjärjestelmien ja palveluiden ohjelmointia varten. Suurin osa pelimoottoreista tarjoaa ohjelmointirajapintoja ohjelmistopaketeissaan. Esimerkiksi Unreal Engine -pelimoottori tarjoaa käyttöliittymän ohjelmoijille pelin luomiseen. Tämä tapahtuu käyttämällä UnrealScript-nimistä komentosarjakieltä sekä kirjastoja, jotka tarjotaan moottorin lisenssin ostaneille. Pakettiin kuuluu muitakin työkaluja, kuten UnrealEd-editori. (Ward 2008.)

Fysiikkamoottori

Fysiikkamoottori on tietokoneohjelma, joka tuottaa virtuaalisen maailman interaktiivisuuden. Ilman sitä pelien oliot eivät liiku tai tunnista törmäystä kohdatessaan. Fysiikkamoottorissa keskitytään tavallisesti vain jäykkien kappaleiden liikkeeseen (kinematiikka/liikeoppi) sekä voimaan ja vääntömomenttiin (dynamiikka), jotka saavat aikaan liikkeen. Fysiikka sekä siitä syntyvät yhteentörmäykset ovat tärkeitä elementtejä melkein kaikissa peleissä. Nykyään harvat peliyritykset kirjoittavat oman fysiikkamoottorinsa. Sen sijaan he käyttävät kolmannen osapuolen valmistamia SDK:ta. Esimerkkinä fysiikkamoottorista voidaan käyttää *Havok Physics* -moottoria. Tämä moottori on niin suosittu peliyritysten käytössä, että sen valinta ymmärretään jo standardina. (Gregory 2009, 41.)

Grafiikkamoottori

Grafiikkamoottori on suurimpia ja monimutkaisimpia pelimoottorin komponentteja. Suurin osa grafiikkamoottoreista rakennetaan laitteistokäyttöliittymäkirjaston, kuten OpenGL:n, päälle. OpenGL on laajasti käytetty laitteistoriippumaton 3D-grafiikkan ohjelmistokehityspaketti. Yksi syy grafiikkamoottorien suureen kokoon on se, että OpenGL:n kaltaiset SDK:t vaativat kohtuullisen paljon koodin kirjoittamista. Grafiikkamoottorista käytetään myös nimitystä *renderöintimoottori*. Alemmalla tasolla renderöintimoottorin eri komponentit keräävät geometristen primitiivien esittelyjä, joita ohjelmoija haluaa saada piirretyksi. Geometrisiä primitiivejä voi olla esimerkiksi viiva- ja pistelistat, partikkelit sekä tekstimerkkijonot. Moottori hallitsee grafiikkalaitteiston tiloja, pelin varjostimia ja piirtää kaikki sille esitellyt geometriat. Piirtämisen rajoittamiseksi tarvitaan korkeamman tason komponentteja, jotka hallitsevat piirtoa ohjelmoijan haluamalla tavalla. (Gregory 2009, 13, 36–37.)

Audio

Audiomoottorit eivät saa paljoa huomiota pelialalla, ja niiden hienostuneisuus vaihtelee laajalti. Esimerkki peruslaatuisesta audiomoottorista löytyy Quak-ja Unreal-pelimoottorien audiomoottorista. Pelinkehittäjät kuitenkin muokkaavat usein valmiin pelimoottorin audiomoottoria tai korvaavat sen kokonaan omalla audiomoottorilla. Jokainen peli vaatii paljon räätälöityjä ohjelmakehityksiä, integrointityötä, hienosäätöä sekä huomiota yksityiskohtiin. Tämä kaikki on tarpeellista luodessa korkealaatuista ääntä lopputuotteeseen. (Gregory 2009, 44.)

Käyttäjäsyötteet

Pelien tarvitsee käsitellä pelaajan antamat syötteet. Nämä syötteet voidaan saada monenlaisista laitteista, HID:stä, jotka toimivat käyttöliittyminä ihmisille (engl. human interface device). Muutamia esimerkkejä näistä laitteista ovat näppäimistö sekä hiiri. HID-moottorikomponentti asetetaan ajoittain moottorin arkkitehtuuriin niin, että se erottaa korkeatasoiset pelikontrollit matalan tason peliohjaimesta. Se käsittelee raa'an datan ohjaimesta tunnistaen nappien tilat sekä muuta. Syötemoottori tarjoaa usein myös mekanismin, joka sallii pelaajan räätälöidä kartoituksen fyysisen kontrollien ja pelilogiikan toimintojen välillä. Se siis antaa pelaajan valita mieluisensa pelikontrollit. (Gregory 2009, 43.)

Tekoäly

Perinteisesti tekoälyn on ajateltu kuuluvan pelispesifisiin kokonaisuuksiin. Se ei ole ollut osana pelimoottoria tai toiminut omana moottorinaan. Peliyritykset ovat kuitenkin huomanneet yhtäläisyyksiä, jotka ilmestyvät jokaisen tekoälyn luonnissa. Jotkin tekoälyn osat ovat näin alkaneet laskeutua moottoripuolen hallinnan alle. Esimerkki tekoälyn SDK:sta on nimeltään Kynapse. Kynapse tarjoaa matalan tason rakennuspalikoita, kuten reitinhaku sekä staattisen ja dynaamisen objektien välttelyn. Muita esimerkkejä SDK:n toiminnoista ovat tilan heikkouksien tunnistaminen sekä tarpeellisen hyvä käyttöliittymä tekoälyn ja animaation välille. Esimerkki tilan heikkouksen tunnistamisesta on avoin ikkuna huoneessa. (Gregory 2009, 33, 47.)

5 KOMENTOSARJAKIELI PELIMOOTTORISSA

Jokainen komentosarjakieli on yksilöllinen kokonaisuutensa, joten niillä on omat menetelmänsä kielen toteutuksesta pelimoottoriin. Tässä luvussa kuvaillaan komentosarjakielien eri rooleja pelimoottorissa. Kohdassa 5.2 keskitytään Lua-kielen toteutuksen suunnitteluun.

5.1 Roolit pelimoottorissa

Komentosarjakielellä kirjoitettu koodi voi toteuttaa monenlaisia rooleja pelimoottorissa. Yksinkertaisimmillaan se voi olla pieni koodin pätkä, joka suorittaa yksinkertaisen funktion olion tai moottorisysteemin puolesta. Toisaalta komentosarjakielillä voidaan luoda korkeatasoisia komentosarjoja, jotka hoitavat koko pelin toimintoja. Gregory (2009) antaa listauksen muutamista mahdollisista arkkitehtuureista, joita komentosarjakielillä voidaan toteuttaa. (Gregory 2009, 802.)

Komentosarjalla kirjoitetut takaisinkutsut

Tässä menettelytavassa pelimoottorin toiminallisuus on laajalti kovakoodattua isäntäkielellä. Jotkin avainosien funktiot on kuitenkin suunniteltu muutettavaksi toiveiden mukaisesti. Tämä on usein toteutettu *hooking*-funktioilla tai takaisinkutsuilla. Nämä ovat käyttäjälle tarjottuja funktiota, jotka moottori kutsuu tarkoituksenaan sallia kustomoinnin. *Hook*-funktiot voidaan kirjoittaa myös natiivikielellä, mutta ne voidaan kirjoittaa myös komentosarjakielellä. Esimerkiksi kun peliolioita päivitetään peliluupin aikana, moottori voi kutsua vaihtoehtoisen takaisinkutsufunktion, joka voidaan kirjoittaa komentosarjana. Tämä antaa käyttäjälle mahdollisuuden kustomoida sitä, miten pelioliot päivittävät itsensä ajon aikana. (Gregory 2009, 802.)

Komentosarjalla kirjoitetut tapahtuman käsittelyt

Tapahtumien käsittelijä on oikeastaan vain erikoistyyppinen *hook*-funktio. Sen tarkoitus on sallia peliolioiden vastata johonkin oleelliseen tapahtumaan pelimaailmassa tai moottorissa itsessään. Esimerkkejä näistä on muun muassa reagoiminen pelissä tapahtuneeseen räjähdykseen tai reagoiminen muistin loppumiseen. Monet pelimoottorit sallivat käyttäjien kirjoittaa tapahtumankäsittelijä-*hook*eja sekä komentosarjana että natiivikielellä. (Gregory 2009, 802.)

Oliotyyppien laajennus tai määrittely komentosarjalla

Jotkin komentosarjakielet sallivat peliolio tyyppejä, jotka on toteutettu natiivikoodilla laajennettavaksi komentosarjojen avulla. Itse asiassa, takaisinkutsut ja tapahtumankäsittelijät ovat pienemmän mittakaavan esimerkkejä tästä. Ajatusta voidaan kuitenkin laajentaa jopa pisteeseen, jossa sallitaan kokonaan uusien pelioliotyyppien määrittely komentosarjassa. Tämä voidaan toteuttaa perinnän tai kokoamisen avulla. Kokoamisella tarkoitetaan komentosarjaluokan instanssin liittämistä natiiviin peliolioon. Perinnässä komentosarjalla kirjoitettu luokka johdetaan natiivi kielellä kirjoitetusta luokasta. (Gregory 2009, 803.)

Komentosarjalla kirjoitetut komponentit tai ominaisuudet

Gregoryn (2009, 803) mukaan komponenttipohjaisessa pelioliomallissa on mielekästä sallia uusien komponenttien oliot rakennettavaksi komentosarjalla. Oliot voidaan rakentaa komentosarjalla joko osittain tai kokonaan. Sama asia pätee myös ominaisuuspohjaisessa pelioliomallissa. Gregory vertaa tätä lähestymistapaa *Gas Powered Games* -nimisen yrityksen peliin *Dungeon Siege*. Peli käytti kyseistä arkkitehtuurimallia. Pelioliomalli oli ominaisuuspohjainen. Sillä oli mahdollista toteuttaa ominaisuuksia joko C++-kielellä tai yrityksen itse kehittämällä kielellä. Kielen nimi on Skrit. Projektin lopussa pelinkehittäjillä oli noin 148 komentosarjalla kirjoitettua ominaisuustyyppiä ja 21 natiivikielellä kirjoitettua ominaisuustyyppiä. (Gregory 2009, 803.)

Komentosarjoilla ohjatut moottori systeemit ja pelit

Komentosarjaa voidaan käyttää ajamaan koko moottorisysteemiä. Esimerkiksi pelioliomallin voisi kuvitella kirjoitettavaksi kokonaan komentosarjalla. Natiivi moottorikoodiin kutsuminen tapahtuisi tällöin vain, kun pelioliomalli vaatii alemman tason moottorikomponenttien toimintoja. (Gregory 2009, 803.)

Jotkin pelimoottorit kääntävät natiivikielen ja komentosarjakielen suhteen päälaelleen. Näissä moottoreissa komentosarjakoodi pyörittää koko toteutusta. Natiivi moottorikoodi toimii vain kirjastona, johon turvaudutaan tarvittaessa moottorin nopeita toimintoja. Gregory (2009, 803) tuo tästä arkkitehtuurityypistä esiin esimerkiksi Panda3D-nimisen moottorin. Panda3D-pelit voidaan kirjoittaa kokonaan Python-kielellä. Natiivi moottori (C++) toimii kuin kirjasto, jota komentosarjakoodi kutsuu. (Gregory 2009, 803.)

5.2 Toteutuksen suunnittelu

Ohjelmoijan näkökulmasta suurin ongelma komentosarjakielen sulauttamisessa peliin on tuottaa sisäänpääsy moottorin olioihin. Kaksi toimintoa, jotka ovat kriittisiä komentosarjakielen valitsemisessa, ovat sulauttamisen ja sidonnan helppous. Yksi syy Luan menestykseen komentosarjakielenä on se, että se on helppo sulauttaa. Luvussa 3 tämä mainittiin yhdeksi Luan suunnitelluksi tavoitteeksi. Lua-kieli ei kuitenkaan tarjoa työvälineitä sidonnan automaattiseen luomiseen. Tämä johtuu siitä, että Luan toinen suunnittelun tavoite on tarjota mekanismi, mutta ei kiveen hakattuja menettelytapoja. Kirjassa *Game Programming Gems 6* (2006) Lua-tiimi kuvaa eri tapoja yhdistää Lua isäntäohjelman olioihin. Seuraavassa taulukossa esitellään koottuja sitomisstrategioita sekä mitä toimintoja kukin niistä tukee. Kappaleen päämääränä on esitellä strategioita peruskirjaston laajentamiseksi sekä C/C++-tyyppisten olioiden käännyttämiseksi. (Ierusalimschy, Figueiredo, Celes 2006, 341–342.)

Jokainen taulukossa 5 esitellyistä strategioista tarjoaa mahdollisuuden päästä raa'asti käsiksi sidottuun olioon komentosarjaympäristöstä. Miettiessä strategiaa Lua-kielen sitomiselle on kuitenkin otettava huomioon kolme asiaa: joustavuus, tehokkuus sekä muistivaatimukset. Olioihin pääsee turvallisesti käsiksi vain, jos strategia suorittaa tyyppitarkistuksen. Olioperusteinen ohjelmointi Luan puolella on mahdollista, kun toteutusstrategia sitoo isäntäoliot komentosarjan olioihin. Taulukon kaksi viimeistä strategiaa sallii sidottujen olioiden käytön as-

sosiaatiotauluina. Tämä toteuttaa laajentumiseen kykenevät oliot. (Kaikissa strategioissa oletetaan, että osoite tunnistaa uniikisti isäntäobjektin.) (Ierusalimschy, Figueiredo, Celes 2006, 353, 346.)

Taulukko 5. Esiteltyjen sitomisstrategioiden tukemat toiminnot (Ierusalimschy, Figueiredo, Celes 2006, 353)

Strategy	**Access**	**Type-Checking**	**OO**	**Extensibility**
Light userdara	✓			
Typed light userdata	✓	✓		
Full userdata	✓	✓	✓	
Extensible userdata	✓	✓	✓	✓
Table	✓	✓	✓	✓

Light userdara

Yksinkertaisin toimintasuunnitelma C/C++-olioiden kiinnittämisestä Luaan on käyttää Lua-kielen ominaisuutta nimeltä *light userdata*. *Light userdata* edustaa osoitinta (void*). Lua kohtelee *light userdataa* tavallisena arvona. Olioiden kartoittaminen yksinkertaisina arvoina tarjoaa kolme etua: yksinkertaisuus, tehokkuus ja pieni muistijälki. Toteutus on yksioikoinen, ja kommunikaatio Luan ja isäntäohjelman välillä on toteutettu tehokkaasti. Se ei näet sisällä epäsuoria muistin varauksia tai pääsyjä olioihin. Tämä strategia ei kuitenkaan ole turvallinen. Tällaisena mikä tahansa userdata-arvo hyväksytään toimivana parametrinä. Epäkelvon olion pääsy voi kaataa koko isäntäohjelman. (Ierusalimschy, Figueiredo, Celes 2006, 344–345.)

Typed light userdata

Ohjelman kaatumisen välttämiseksi Lua-ympäristössä toteutusstrategiaan lisätään ajon aikana toimiva tyypin tarkistus. Tyyppitarkistuksen lisääminen vähentää suorituskykyä ja kasvattaa muistin käsittelyä. Tyyppitarkistuksen lisäämiseksi on luotava taulu, jossa laitetaan pareiksi jokaisen Luaan kartoitetun olion osoite ja sitä vastaava tyyppinimi. Taulu käyttää *light userdataa* avaimina ja tyyppien nimiä arvoina. (taulu[osoite] = ”nimi”) Tauluun ei saa kuitenkaan päästä käsiksi Lua-ympäristön puolelta. Isäntäohjelma voi muuten kaatua Lua-komentosarjasta. Tämä voidaan estää käyttämällä Luan *environment table* -ominaisuutta, jota

käytetään C-funktioiden puolella. Tyyppi-tarkistustaulu asetetaan sitomisfunktioiden *environment*-tauluksi. Funktioiden sisällä voidaan päästä tehokkaasti käsiksi tauluun. (Ierusalimschy, Figueiredo, Celes 2006, 345–346.)

Tämä strategia on vielä hyvin tehokas. Myös muistirasite on aika alhainen. Tämä strategia on kuitenkin hyvin rajattu, sillä se ei salli olioperusteista ohjelmointia. Koska isäntäolioiden sidonta toteutetaan käyttämällä *light userdataa*, isäntäoliot kartoitetaan arvoina. Näin ollen ei voida käyttää olio-ohjelmoinnin syntaksia metodien kutsumiseen eikä Lua kykene hallitsemaan olioiden elinikää. (Ierusalimschy, Figueiredo, Celes 2006, 347.)

Full userdata

Isäntäolioiden käsittelemiseksi Lua-olioina on käytettävä ominaisuutta nimeltä *full userdata*. Luassa *light userdataa* ja *full userdataa* ei pystytä erottamaan toisistaan. *Full userdatan* kanssa isäntäohjelma voi kuitenkin käyttää metatauluja, joilla määritellä tiettyjä käyttäytymisiä. Tässä strategiassa asetetaan erillinen metataulu jokaiselle erilliselle oliotyypille. Samantyyppiset oliot jakavat siis saman metataulun. Apukirjasto lisää rekisteritauluun merkinnän jokaisesta luodusta metataulusta. Näin kartoitetaan tyyppinimi metatauluun. Annettaessa koodissa haluttu oliotyyppi voidaan päästä käsiksi vastaavaan metatauluun. (Ierusalimschy, Figueiredo, Celes 2006, 347.)

Olioiden sitominen Lua olioina tuo monia etuja. Isäntäolioita käsitellään Lua-olioina. Käyttämällä samaa syntaksia strategia turvautuu Luan automaattiseen roskien keräilijään vapauttaakseen varatun muistin. Lisäksi suoritus hidastuu huomattavasti vain, kun olioiden kartoitus tapahtuu ensimmäisen kerran. Perinnän kyky pohjaluokasta havainnollistaa myös etua, joka tulee sidotessa isäntäoliot Lua-olioihin. Tämä ei olisi mahdollista, jos olioita kohdeltaisiin tavallisina arvoina. (Ierusalimschy, Figueiredo, Celes 2006, 349, 351.)

Extensible userdata

Olio-ohjelmoinnin käyttöönoton lisäksi on mahdollista tehdä olioista laajennettavia. Laajentaminen voi tarkoittaa uusien ominaisuuksien liittämistä olioon, olemassa olevien metodien uudelleen määrittelyä tai laajentaa olion käytöstä. Tämä on mahdollista assosiaatiotaulujen ansiosta. Assosiaatiotaulut voidaan luetteloida millä tahansa arvolla (paitsi *nil*). Strategian ta-

voitteena on siis kohdella isäntäolioita assosiaatiotauluina. Tämän strategian yksi varjopuoli on se, että suorituskyvystä sakotetaan, kun yritetään päästä käsiksi isäntämetodeihin. (Ierusalimschy, Figueiredo, Celes 2006, 351–352.)

Table

Kun ohjelman tarvitsee usein kiinnittää lisäalueita, on olemassa yksinkertaisempi strategia sitoa isäntäoliot. Tämä strategia sitoo isäntäoliot Lua-tauluihin. Sen sijaan, että luotaisiin uusi *full userdata* jokaiselle oliolle, voidaan luoda taulu jokaiselle oliolle. Kaikki userdatan ominaisuudet voidaan soveltaa tauluihin. Näitä ominaisuuksia olivat olioperusteinen syntaksi, automaattinen roskien keräys sekä lisäominaisuuksien varastointi. Verratessa *extensible userdata* -strategiaan tämän lähestymistavan ainoa haittapuoli on se, että se varastoi enemmän muistia olion sitomiseen. Tämä tapahtuu, mikäli strategiassa ei käytetä hyväksi sen lisäominaisuuksia. Lisämuisti, joka on varattu tälle toiminnalle, varastoituu siis turhaan. Taulun muistijälki on suurempi kuin yhden userdata pointerin muistijälki. (Ierusalimschy, Figueiredo, Celes 2006, 352–353.)

5.3 Komentosarjakielen lisäys

Komentosarjakielen lisäystyyli voi olla riippuvainen komentosarjakielen rakenteesta. Myös kielen kehittäjillä voi olla oma näkemys siitä, miten kielen lisäys olisi parhaiten toteutettava. Tässä kappaleessa keskitytään komentosarjakielen lisäykseen Lua-kielen näkökulmasta. Luan virallisen nettisivun dokumentaatiossa kuvataan lyhyesti mitä tehdä, jos Lua-kieltä halutaan käyttää ei-Unix-työkaluilla. Ohjeistuksen kerrotaan riippuvan siitä, mitä kääntäjää halutaan käyttää. Lua-kirjaston, tulkitsijan ja kääntäjän oikean tietokoneohjelman version tekemiseksi tarvitaan luoda taulukon 6 mukaiset projektit. (LUA 2014 c.)

Taulukko 6. Lua-tiedostot (LUA 2014 c)

Kirjasto	lapi.c, lcode.c, lctype.c, ldebug.c, ldo.c, ldump.c, lfunc.c, lgc.c, llex.c, lmem.c, lobject.c, lopcodes.c, lparser.c, lstate.c, lstring.c, ltable.c, ltm.c, lundump.c, lvm.c, lzio.c, lauxlib.c, lbaselib.c, lbitlib.c, lcorolib.c, ldblib.c, liolib.c, lmathlib.c, loslib.c, lstrlib.c, ltablib.c, loadlib.c, linit.c
Tulkitsija	Kirjasto, lua.c
Kääntäjä	Kirjasto, luac.c

Dokumentaatiossa kerrotaan, että Lua-kirjaston käyttämiseksi ohjelmoijan on osattava luoda ja käyttää kirjastoja valitsemallaan ohjelmointikielen kääntäjällä. Windows käyttöjärjestelmän käyttäjille suositellaan dynaamisesti linkitettävän kirjaston muotoa. Lisäksi C-kirjastojen lataamiseksi dynaamisesti tarvitaan tietotaito dynaamisten kirjastojen luomisesta. Lua-ohjelmointirajapintafunktioiden on myös päästävät käsiksi dynaamisiin kirjastoihin. (Dokumentaatio kuitenkin varoittaa olla linkittämättä Lua-kirjastoa jokaiseen käytettävään dynaamiseen kirjastoon.) (LUA 2014 c.)

Projektiin pudottaminen

Helpoin tapa Lua-kielen käyttöönottoon on lisätä ladatut lähdekooditiedostot sellaisenaan projektitiedostoon. Pakatun lähdekoodin lataamisen ja paketin purkamisen jälkeen Luan tiedostot voi lisätä samaan tiedostopolun sijaintiin kuin tavalliset Visual Studio -projektin C++-tiedostot. Tämän jälkeen tiedostot lisätään Visual Studiossa projektiin. Tiedostot voidaan lisätä niin kuin ohjelmoija itse parhaakseen näkee. Luan lähdekoodin *.h*-ja *.c* -tiedostot voidaan lisätä Visual Studion tarjoamiin *Header Files*-ja *Source Files* -kansioihin muiden Visual Studiossa luotujen tiedostojen joukkoon. Vaihtoehtoisesti tiedostot voidaan asettaa erilliseen itse luotuun *Lua*-kansioon. Tällöin projektin hallitseminen voidaan kokea helpommaksi. Lua-lähdekoodi löytyy tarvittaessa omasta kansiosta ja projektikeskeinen koodi omastaan. (LuaBinaries 2014 a; TECHNEILLOGY 2012.)

Lähdekoodin lisäyksessä on muistettava ottaa huomioon kaksi c-tyyppistä tiedosta. Tiedostot *lua.c* ja *luac.c* ovat tiedostoja, jotka molemmat sisältävät koodissaan *main()*-osion. Ensimmäinen tiedosto on Lua tiedostotulkitsija. Jälkimmäinen on tavukoodikääntäjä. Riippuen siitä, miten ohjelmoija haluaa käsitellä koodinsa kääntämisen, toinen tai molemmat tiedostot poistetaan. Ohjelmoija voi myös kirjoittaa oman tiedoston kääntäjän. (LuaBinaries 2014 a; TECHNEILLOGY 2012.)

Staattinen kirjasto

Staattisen Lua-kirjaston luomiseksi on luotava erillinen projekti Visual Studiossa. Tätä kutsutaan staattiseksi kirjastoprojektiksi. Staattinen kirjasto tulee osaksi ohjelmakoodia. Se vähentää ohjelman riippuvuuksia, mutta samalla kasvattaa ohjelman kokoa. Luan virallisen nettisivun dokumentaatio osio suosittelee staattisen kirjaston käyttämistä Unix käyttöjärjestelmällä (LUA 2014 c). (Developer Network 2014 a.)

Dynaamisesti linkitettävä kirjasto

Dynaamisesti linkitettävä kirjasto (engl. dynamic-link library, DLL) on exe-tiedosto, joka toimii funktioiden yhteisenä kirjastona. Dynaaminen linkitys tarjoaa tavan käsitellä funktiokutsuja, jotka eivät ole osana konekielistä koodia. Konekielinen koodi funktioille sijaitsee DLL:ssä. DLL sisältää yhden tai useamman funktion, jotka on käännetty, linkitetty ja tallennettu erilliseksi prosessista, joka käyttää niitä. Dynaamisesti linkitettävät kirjastot myös johtavat datan ja resurssien jakoa. Useat sovellutukset voivat päästä käsiksi samanaikaisesti yhden DLL-kopion sisältöön muistissa. (Developer Network 2014 b.)

Dynaaminen linkitys poikkeaa staattisesta linkityksestä siten, että se sallii exe-tiedoston moduulin (.dll tai .exe) sisältää vain ajon aikana tarvittavaa tietoa konekielisen koodin paikantamiseen DLL-funktiosta. Staattisessa linkityksessä linkkeri saa kaikki viitatut funktiot staattisesti linkitetystä kirjastosta ja asentaa sen kirjoitetun koodin kanssa ajotiedostoon. Käyttämällä dynaamista linkitystä staattisen linkityksen sijaan saadaan käyttöön monia hyviä puolia. Dynaamisesti linkitettävät kirjastot muun muassa säästävät muistia, ovat helpompi päivittää, tukevat monikielisiä ohjelmia ja helpottavat kansainvälisten versioiden luomista. Jotkin Lua-kielen funktioiden toiminta on riippuvainen kirjaston muodosta (LUA 2014 c). (Developer Network 2014 b.)

6 TESTIPROJEKTIN TOTEUTUSPROSESSI

Tässä luvussa käsitellään opinnäytetyön käytännön osuutta, sen aikana tehtyjä valintoja ja niiden syitä. Käytännön osuudessa toteutetaan testiprojekti, jossa tehdään Luan toteutus valitulle pelimoottorille. Luvussa viitataan käytännön osuuden aikana käytettyihin ohjeistuksiin ja lähteisiin. Ohjeistukset ovat suurimmilta osin eri Lua-ohjelmoijien itse kirjoittamia kokemuksia ja neuvoja. Tekstissä on pyritty heijastamaan näitä ohjeistuksia niihin virallisiin julkaisuihin, joita Lua-kielen kehittäjät ovat tuottaneet.

6.1 Tavoitteet

Käytännön osuuden tavoitteena on asentaa Lua-komentosarjakieli valitulle moottorialustalle ja toteuttaa sen toimintoja kyseisellä alustalla. Työ toteutetaan käyttämällä Microsoft Visual Express C++ -ohjelmaa sekä Luan versiota 5.2. Opinnäytetyön teon käyttöjärjestelmänä on Windows 7 -ympäristö. Asentamisen jälkeen kieli sulautetaan moottoriin. Moottorista valitaan jokin luokkaolio, jonka toimintoja laajennetaan Lua-kieltä käyttäen. Tämä voi tarkoittaa esimerkiksi C++-luokan muokkaamista pohjaluokaksi niin, että siitä voidaan johtaa erilaisia olioita Lua-komentosarjojen avulla.

Testiprojektin tarkemmaksi aiheeksi päätettiin luoda 3D-pallo, joka liikkuu rajatun alueen sisällä. Jos pallo osuu alueen reunoihin, se kimpoaa toiseen suuntaan. Testiprojektin viimeisessä vaiheessa yritetään luoda rajattuun tilaan useita palloja, jotka kimpoavat eri suuntiin. Tällöin pallojen pitää myös reagoida osuessaan toisiinsa. Mikäli ylimääräistä aikaa riittää testiprojektin tekemiseen, pallot ohjelmoidaan katoamaan osuessaan toisiinsa valitun lukumäärän jälkeen. Pallon tiedot ja sen tarvitsemat siirto- ja tarkastusmetodit kirjoitetaan Lua-komentosarjoilla. Nämä olivat suunnitelmani opinnäytetyön käytännön osuuden tekemiseen.

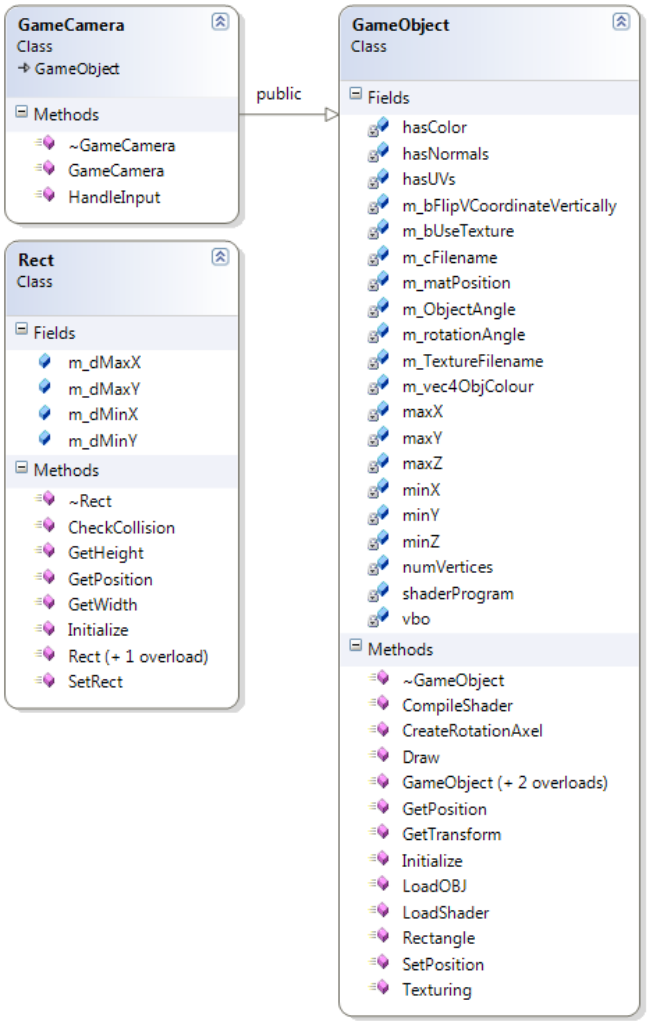
6.2 Pelimoottorin valinta

Testiprojektiin valittiin käytettäväksi ohjatusti kirjoittamani moottori. Moottori toteutettiin vuonna 2013 Kajaanin ammattikorkeakoulun Pelimoottoriprojekti II -kurssilla. Kurssin oh-

jaajana toimi Jukka Jylänki, joka tuotti kurssille joitakin tarvittavia kirjastoja sekä virheiden jäljittäjäluokkia moottorin toteuttamiseen. Kurssissa keskityttiin 3D-grafiikkamoottorin toteutukseen sekä pelimoottorin perusrakenteeseen. Vaikka kurssilla toteuttamani lopputulos on tasoltaan vain tyydyttävä, tämä moottori valittiin opinnäytetyön testiprojektiin pienen kokonsa vuoksi. Vaihtoehtoisesti valittavana olisi ollut kolmannen osapuolen toteuttama moottori. Opinnäytetyössä haluttiin kuitenkin säästää aikaa uuden moottorin oppimisesta. Luan lisääminen pelimoottoriin oli myös ollut ennen aikaisena aikomuksenani Pelimoottoriprojekti II -kurssin aikana. Kielen lisääminen jäi kuitenkin toteuttamatta tarvittavan ajan sekä sen aikaisen osaamisen puutteessa.

Kuva 2 näyttää moottorissa toteutetut luokat. Visual Studion automaattinen luokkahierarkian piirtäjä ei tunnista kaikkia moottorissa olevia tiedostoja, kuten *Engine.h* ja *Engine.cpp*. Jälkimmäinen tiedosto sisältää moottorin *main()*-tiedoston sekä sen piirto-ja päivitysfunktiot. *Engine.h*-tiedosto esittelee joitakin moottorille olennaisia muuttujia. Koska moottorin alemman tason toiminnat, kuten matriisilaskennat, eivät kuulu opinnäytetyön aihepiiriin, työstä käsitellään vain komentosarjan kirjoittamisen sulauttamiseen liittyviä moottorin osia ja luokkia

Moottorilla voidaan luoda ja ladata 3D-hahmoja sekä tekemään yksinkertaista kaksiulotteista yhteentörmäyksen tunnistamista z-akselista katsottuna. Moottorin *GameObject*-olioluokasta on johdettu kameraluokka, joka mahdollistaa ladattujen 3D-objektien tarkastelemisen moottorin luomasta ikkunasta. Opinnäytetyön käytännön osuuden tavoitteena on saada Lua-kieli kommunikoimaan näiden luokkien kanssa.



Kuva 2. Moottorin luokkahierarkia (Visual Studio 2010)

6.3 Luan lisäämisen suunnittelu

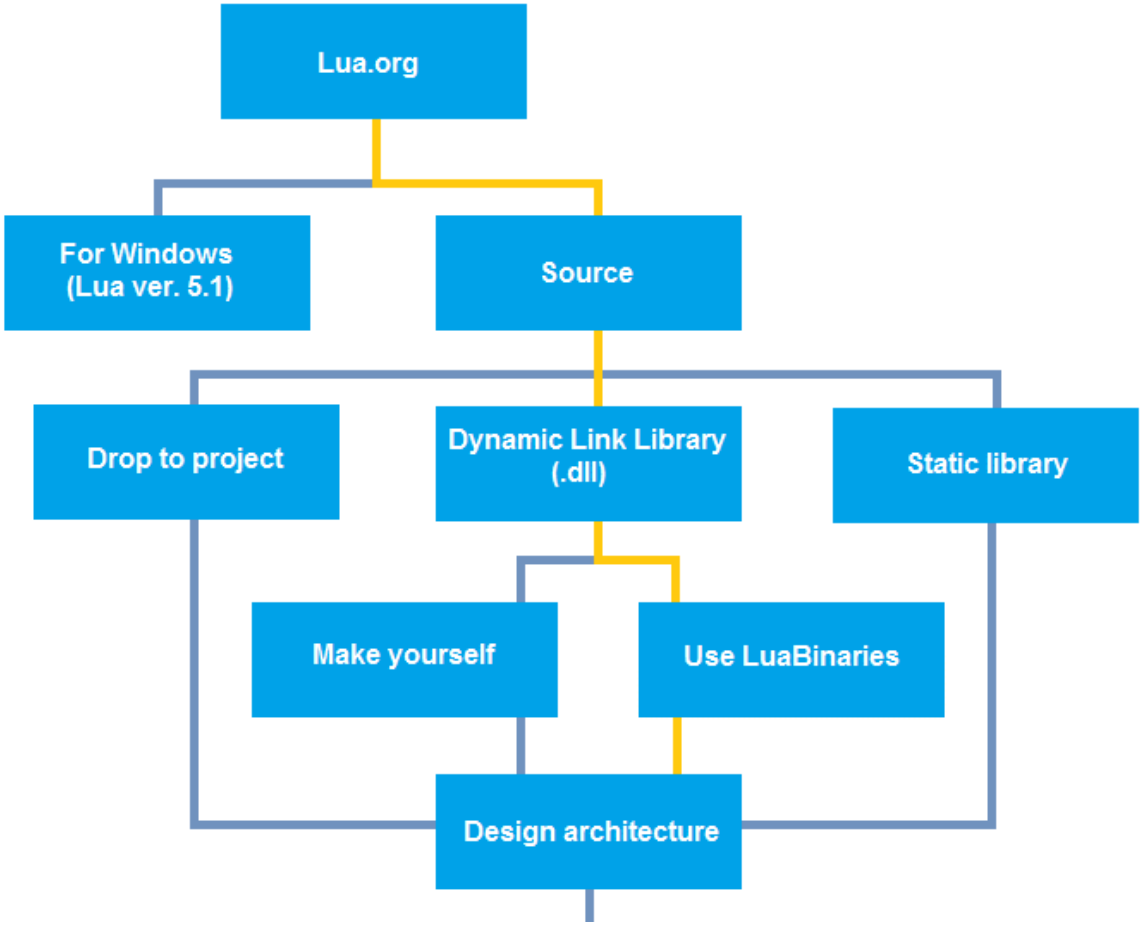
Luan lisäämisen suunnittelun ensimmäisenä osana oli päättää miten yhdistää Luan kirjasto ja virtuaalikone projektiin. Lua-kielen virallisilla nettisivuilla mainitaan nimellisesti Lua-kielen asennusohjeet Microsoft Windows-, Mac OS X-ja Linux-käyttöjärjestelmille. Luan sivuilla kerrotaan (2014), että Linux ja Mac OS X -käyttöjärjestelmistä Lua-kielen voi löytää jo asennettuna tai sille on olemassa paketti. Lua-kielen viralliset nettisivut antavat näille kahdelle myös yksityiskohtaiset ohjeet Luan rakentamiseen ja asentamiseen. Windows-käyttöjärjestelmälle asia on monimutkaisempi. (LUA 2014 b.)

Luan asentamisesta Visual Studio -ympäristöön ei löydy paljoa ohjeistusta. Muutamat kirjoitelmat, jotka tämän opinnäytetyön teon yhteydessä ovat löytyneet, ilmaisivat samasta ongelmasta (Johnson 2013; Dunbar 2013). Luan viralliset nettisivut tarjoavat linkin Luan-wikiin, jonka tiedot kuitenkin osoittautuivat Lua-versiostaan vanhentuneeksi. Ohjeistukset ja ehdotukset Luan virallisella sivulla ovat myös vaihtuneet opinnäytetyön aloittamisen jälkeen. Ensimmäinen ehdotus Windows-käyttöjärjestelmän käyttäjille oli ladata *LuaForWindows*-niminen valmis ympäristö. Tämä ehdotus hylättiin kuitenkin opinnäytetyön käytöstä, koska *LuaForWindows* on toteutettu Luan 5.1-versiolla. Lisäksi valmiin ympäristön käyttäminen Visual C++ kanssa olisi monimutkaisempaa kuin Luan lisääminen suoraan Visual C++ -ympäristöön. Linkki *LuaForWindows*-ympäristön lataamiseen on sittemmin poistettu Luan viralliselta *Getting Started* -sivulta ja sen tilalle on annettu ehdotukseksi käyttää *LuaDist*-nimistä monialustapakettia. (LUA 2014 b; LuaForWindows 2013.)

Opinnäytetyössä halutaan sulauttaa Lua-kieli pelimoottoriin. Näin ollen jäljelle jää lähdekoodin suora käsittely. Luan asentamiseksi voidaan ladata paketti, joka on käännetty käytettävälle käyttöjärjestelmälle, tai ladata lähdekoodi ja kääntää se itse (LUA 2014 b). Luan virallisen nettisivun dokumentaatio suosittelee Lua-kirjaston käyttämistä dynaamisena linkitettynä kirjastona (LUA 2014 c). Ennen opinnäytetyön aloittamista en ole ennen tehnyt itse dynaamisesti linkitettyä kirjastoa. Netistä löytyneet ohjeistukset kuvailivat, miten prosessi tulisi toteuttaa. Ajatus mahdollisista virheistä, jotka voisivat tapahtua virheellisesti toteutetun prosessin takia sekä niiden korjaamiseen kuluva aika saivat minut kuitenkin tarkastelemaan mahdollisia valmiiksi käännettyjä dll-tyyppisiä kirjastoja.

Etsimäni löytyi LuaBinaries-nimiseltä nettisivulta, jonka tarkoituksena on vähentää epäyhdenmukaisuuksien tapahtumista luotujen Lua-kirjastojen välillä. Sivuston binääritiedostojen eri tietokoneohjelman versiot on kääntänyt ja ylläpitänyt Tecgrafin Antonio Scuri. Projektin suunnittelivat André Carregal sekä Antonio Scuri, Danilo Tulerin ja Diego Nehabin avustuksella. (LuaBinaries 2014 b.)

Kuva 3 esittelee Luan lisäämisen suunnittelussa tapahtuneita vaihtoehtoja. Kirkastetut polut kuvaavat valittuja vaihtoehtoja. Kuvan tarkoituksena on hahmottaa sitä mahdollisuuksien määrää, joilla voidaan vaikuttaa Luan käytön toteutukseen ennen kuin yhtään komentosarjaa on mahdollista kirjoittaa Lualla.

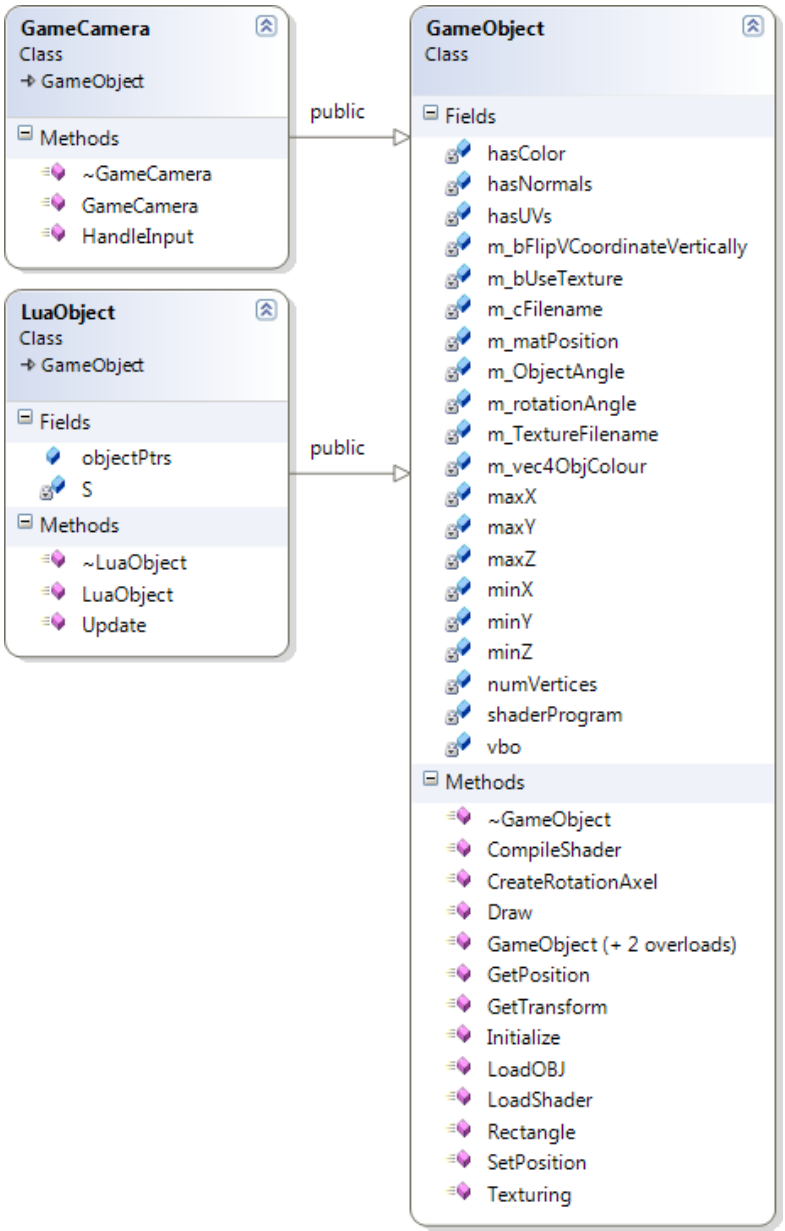


Kuva 3. Luan lisäämisen suunnittelu

6.4 Luan toteutuksen suunnittelu

Lua-kielen kirjaston lisäämistyylin jälkeen oli suunniteltava, miten varsinainen Lua-kielen käytön toteutus tapahtuu moottorissa. Idean toteuttamiseksi minun oli opeteltava, miten C++ ja Lua voivat kommunikoida keskenään. Tarkastelin useita erilaisia netistä ja ohjelmointikirjoista löydettäviä ohjeistuksia, jotka auttoivat minua oppimaan asiasta. Ensimmäinen versio testiprojektin toteutuksesta oli tilkkutäkkimäinen kokoelma kyseisten ohjeistuksen tyyleistä. Esimerkkien muokkaaminen omaan tarkoitukseeni sekä koodin päivittäminen Lua-kielen 5.2-versioon muovasivat kuitenkin toteutuksen lopulta omaksi kokonaisuudekseen.

Toteutuksen aloittelevana ajatuksena oli luoda laajennus moottorin *GameObject*-luokkaan. Halusin antaa moottorille erillisen olion, joka hallitsisi Lua-komentosarjan kautta luotuja olioita. Moottorin puolelle luotiin luokka *LuaObject*. Kuva 4 näyttää *LuaObject*-luokan aseman luokkahierarkiassa. *LuaObject*, joka on johdettu *GameObject*-luokasta, saa tarvittavat 3D-objektin piirron ja hallinnan ominaisuudet. Lisäksi se saa omaksi muuttujakseen lua_State-tyypin, jonka avulla päästään käsiksi Lua-pinoon.



Kuva 4. LuaObjekti-luokan asema luokkahierarkiassa (Visual Studio 2010)

Pelkkä C++-olio, joka pitää sisällään lua_State-tyyppistä muuttujaa, ei kuitenkaan riitä sitomisprosessin toteuttamiseen. Sitomisen toteuttamiseksi Lua-kielen on päästävä käsiksi C++-moottorin ominaisuuksiin. Tämä tapahtuu luomalla moottoriin funktioita, jotka rekisteröi-

dään Lua-kielelle käytettäväksi. Suurin osa toteutuksen C++- ja Lua-kielten välisestä kommunikaatiosta tapahtuu näissä rekisteröitävissä funktioissa. Moottorin puolelle tarvitaan myös cpp-tiedosto, jossa kielen sitominen tapahtuu. Luan käyttämien C++-funktioiden sitomista varten luotiin cpp-tiedosto *TestCase* (LIITE 2).

TestCase.cpp sisältää kolme rekisteröitävää funktiota. Näitä ovat 3D-objektin luomisen toteuttava funktio LUA_NewLuaObject, Lua-komentosarjan päivitysfunktion asetusfunktio LUA_SetUpdate sekä LUA_MoveObject, joka asettaa komentosarjalta saadun sijainnin moottorin piirtoon. Jokainen rekisteröitävä funktio tarvitsee parametrikseen lua_State-tyyppisen osoittimen. Funktioiden on myös palautettava kokonaislukuarvo, joka kertoo, kuinka monta osaa Lua-pinosta palautetaan takaisin Lua-komentosarjalle. Viimeinen liitteessä 2 oleva init-funktio toteuttaa funktioiden rekisteröinnin. Init-funktio kutsutaan moottorin *main.cpp*-tiedostossa (LIITE 3), joka toteuttaa kaikki ennen ohjelman aloittamista tarvittavat asennukset.

Lua-arvon tallennus C-koodissa

Testiprojektissa halutaan pystyä luomaan erilaisia 3D-objekteja *LuaObject*-luokan ja Lua-komentosarjojen avulla. Olioiden tiedot kirjoitetaan Lua-komentosarjoilla, mutta *LuaObject*-luokka kutsuu Luan tiedot moottorissa. Uniikin päivitysfunktion toteuttamiseksi luotu olio on yhdistettävä sen omaan päivitysfunktiokomentosarjaan. Tähän tarkoitukseen käytetään Luan C-API rekisteritaulua. Se on ennalta määrätty taulu, johon C-koodi voi tallentaa mitä tahansa Lua-arvoja. Rekisteritaulu sijaitsee Luan sovellusohjelmointirajapinnan pseudoindeksissä LUA_REGISTRYINDEX. Ristiriitojen välttämiseksi taulun avaimina käytetään luotujen olioiden osoittimia *light userdatana*. Taulukko 7 havainnollistaa Lua-komentosarjan tallennettavan funktion (taulukossa *function*) sijaintia rekisteritaulussa. (LUA 2014 d.)

Taulukko 7. Lua-komentosarjafunktion sijainti rekisteritaulussa

Taulu	Avain	Arvo
LUA_REGISTRYINDEX	[light userdata]	table[”update”] = function

Userdata edustaa C-arvoja Luassa. *Light userdata* edustaa void-tyyppistä osoitinta, void*. Se on numeron kaltainen arvo. *Light userdata* on samanarvoinen kuin mikä tahansa *light userdata*, jolla on sama C-osoite. (LUA 2014 d.)

Rekisteritaulu saa osoitinavaimensa (taulussa *light userdata*) uuden objektin luonnin yhteydessä funktiossa LUA_NewLuaObject. Samalla LUA_REGISTRYINDEX saa annetun avaimen paikalle arvokseen tyhjän taulun (*table*). Tämä tyhjä taulu saa oman avaimen ”update” LUA_SetUpdate-funktiossa. *Table*-taulun arvoksi annetaan Lua-komentosarjan antama päivitysfunktio (*function*).

6.5 Käytännön toteutus

Käytännön toteutus koostuu toivotun kokonaisuuden erillisten osien rakentamisesta siinä järjestyksessä kuin niiden teko on tuntunut taitoihin sopivimmalta. Käytännön osuus on jaettu 3D-objektin luontiin, sen sijainnin muokkaukseen sekä Lua-komentosarjan kautta rakennetun päivitysfunktion asettamiseen ja kutsumiseen. Usein toistuvat ohjelmoinnin menetelmät kuvaillaan tarkasti vain ensi kerran käytettäessä. Koodin kuvauksissa osoittimiin viitataan niiden nimellä tai tyypillä varustettuna asteriksimerkillä.

LuaObject

Moottoriin luotiin *GameObject*-luokasta johdettu luokka *LuaObject*. Luokan tarkoitus on mahdollistaa erilaisten olioiden ja 3D-objektien luonti komentosarjoilla. Jotta olio voi päästä käsiksi Lua-tilan Lua-pinoon, luokkaan lisätään lua_State-tyyppinen muuttujaosoitin *S*. Luokalle luotiin myös liikkeen päivityksen yhteydessä staattinen vektorilista *objectPtrs*. Koska testiprojektissa halutaan pystyä luomaan useita objekteja, tämä lista tulee tarpeelliseksi. *LuaObject*-luokka saa muodostimeensa parametriksi lua_State*-tyypin. Muodostin näyttää *LuaObject.cpp*-tiedostossa seuraavalta:

```
LuaObject::LuaObject(std::string VertexFilename,
                     bool CoordinateVertically,
                     std::string TextureFilename,
                     lua_State* state)
       :GameObject(VertexFilename, CoordinateVertically, TextureFilename),
       S(state)
{
       objectPtrs.push_back(this);
}
```

Kun uusi olio luodaan koodissa, muuttuja *S* alustetaan muodostimessa ja *objectPtrs* vektorilistaan lisätään viittaus luodusta objektista. Vektorilista esitellään moottorin *GameVariables.h*-

tiedostossa, ja sitä käytetään moottorin *main.cpp*-tiedoston (liite 3) UpdateLoop päivitysluupissa. Vektorilista käydään for-luupissa läpi. Jokainen vektorilistaan lisätty olio kutsuu vuorollaan omaa Update-metodiaan. Kuva 5 näyttää *LuaObject*-luokan Update-metodin toiminnan.

```
void LuaObject::Update()
{
    //Pushes a light userdata onto the stack
    lua_pushlightuserdata(S, this);

    //Pushes onto the stack the value of LUA_REGISTRYINDEX[lightuserdata]
    lua_gettable(S, LUA_REGISTRYINDEX);

    //Pushes onto the stack the value of table["update"]
    lua_getfield(S, -1, "update");

    //Calls a function
    lua_call(S, 0, 0);
}
```

Kuva 5. LuaObject-luokan Update-metodi (Visual Studio 2010)

Update-metodissa kutsutaan Lua-tauluun kiinnitetty päivitysfunktio. Funktion kutsumiseksi se on ensin tuotava Lua-tilan Lua-pinoon. Koska funktio on osa rekisteritaulun arvoa, Lua-pinoon on ensin työnnettävä olion *light userdata*, joka toimii rekisteritaulun avaimena. Rekisteritaulun arvon ollessa Lua-pinossa voidaan pinoon työntää ”update”-avaimen pitämä arvo. Tämän jälkeen Update-metodi voi kutsua komentosarjalla kirjoitetun päivitysfunktion.

3D-objektin luonti

Käytännön osuudessa ensimmäisenä rekisteröity funktio oli LUA_NewLuaObject. Luan ja C++-kielten kommunikointi haluttiin aloittaa yksinkertaisella annettujen parametrien siirrolla Lua-komentosarjasta C++-kielen puolelle. Tarkoituksena oli saada tarvittavat parametrit 3D-mallin lataamiseen ja piirtoon. Kun tämä onnistui, funktioon lisättiin rekisteritaulun avaimen lisäys. Kuva 6 havainnollistaa LUA_NewLuaObject-funktion toimintaa.

```
int LUA_NewLuaObject(lua_State *S)
{
    //Checks and takes parameters from Lua stack
    luaL_checktype(S, -1, LUA_TSTRING);
    const std::string textureFilename = lua_tostring(S, -1);
    luaL_checktype(S, -2, LUA_TSTRING);
    const std::string vertexFilename = lua_tostring(S, -2);

    //Allocates a new block of memory with the given size,
    //pushes onto the stack a new full userdata with the block address
    LuaObject *newObject = (LuaObject*)lua_newuserdata(S, sizeof(LuaObject));
    new (newObject)LuaObject(vertexFilename, true, textureFilename, S);

    lua_pushlightuserdata(S, newObject);
    lua_newtable(S);
    lua_settable(S, LUA_REGISTRYINDEX);

    //Adds object to GameObject vector
    gamePointerObjects.push_back((GameObject*)newObject);

    return 1;
}
```

Kuva 6. Viimeinen versio LUA_NewLuaObject-metodista (Visual Studio 2010)

Lua-tilan Lua-pinosta otetaan komentosarjan antamat parametrit ja asetetaan ne *LuaObject*-luokan muodostimeen. Funktiolla lua_newuserdata varataan *LuaObjectin* verran muistia ja työnnetään Lua-pinoon uusi *full userdata* muistipaikan lohkon (engl. block address) kanssa. Toiminta palauttaa tämän viittauksen. Moottori voi käyttää tätä varattua muistia. Muodostin asetetaan varattuun muistiin käyttämällä new-operaattoria ja viittaamalla sen varauksessa käytettyyn osoitinmuuttujaan. Kun olio on luotu, se lisätään *GameObject*-tyyppisenä osoittimena moottorin *gamePointerObjects*-listaan. Listalla käydään läpi moottorissa piirrettävät oliot. (LUA 2014 d.)

Rekisteritaulun paikan asettamiseksi Lua-pinoon työnnetään avaimena toimiva *light userdata* sekä arvoksi tuleva tyhjä taulu. Funktio lua_settable ottaa pinoon työnnetyt arvot ja asettaa ne rekisteritauluun. Funktiokutsu poistaa Lua-pinosta *light userdatan* sekä tyhjän taulun. Viittauksen pitämiseksi tallessa Lua-komentosarjan puolella on luotava taulu, joka ottaa vastaan palautetun arvon (liite 1 3(3)). Taulua käytetään viitattaessa luotuun objektiin. (LUA 2014 d.)

3D-objektin siirtäminen

Käytännön osuuden seuraavaksi osaksi haluttiin toteuttaa luodulle pallolle sijainnin siirtäminen. Tätä varten toteutettiin rekisteröitävä funktio, joka kertoo moottorille sen piirtämän objektin sijainnin muutoksen. LUA_MoveObject (kuva 7) ottaa Lua-pinosta parametreinä annetut sijaintiarvot ja tarkistaa niiden tyypin. Jos annetut tyypit ovat numeroarvoja, ne

asennetaan *GameObject*-luokan metodin SetPosition parametriarvoiksi. Koska koodiin tai komentosarjoihin ei ole tässä vaiheessa luotu viittausmenetelmiä luotujen objektien tunnistamiseksi ja tiedetään, että luotuja objekteja on vain yksi, viitataan funktiossa *gamePointerObjects*-listan ainoaan luotuun objektiin.

```
int LUA_MoveObject(lua_State *S)
{
    luaL_checktype(S, -1, LUA_TNUMBER);
    const float yPos = lua_tointeger(S, -1);
    luaL_checktype(S, -2, LUA_TNUMBER);
    const float xPos = lua_tointeger(S, -2);

    //Set new position to object in list
    gamePointerObjects.at(0)->SetPosition(xPos, yPos, 0.0f);

    return 0;
}
```

Kuva 7. Moottorin TestCase.cpp:n LUA_MoveObject-funktio (Visual Studio 2010)

Uuden funktion testaamiseksi *main.lua*-tiedostoon kirjoitetaan objektin luonnin perään MoveObject funktio. Parametreiksi annetaan halutut x- ja y-akseleiden mukaiset arvot. Kun koodi käännetään, luotu pallo on siirtynyt eri paikkaan aikaisemmasta sijainnistaan.

Päivityksen asettaminen

Tähän mennessä Lualla pystytään luomaan 3D-objekti sekä siirtämään sen sijaintia. Jatkuvan liikkeen päivityksen toteuttamiseksi rekisteritauluun on asetettava komentosarjana kirjoitettu päivitysfunktio. Funktio LUA_SetUpdate (kuva 8) toteuttaa arvon asettamisen rekisteritauluun. Funktion alussa Lua-pinossa on komentosarjan puolella parametreinä annettu olion taulu sekä olioon lisättävä päivitysfunktio. Taulu on pinon pohjalla ja funktio pinon päällimmäisenä arvona. Liitteessä 1 on Lua-komentosarjan puoleinen toteutus.

Olion taulu annetaan parametrinä, koska siitä saadaan tarvittava muistiosoite (engl. block address) rekisteritaulun avaimelle. Kun rekisteritaulun arvo on kutsuttu oikealla avaimella, sille annetaan oma avain ja arvo. Avaimeksi tyhjälle taululle annetaan merkkijono ”update”. Avainpaikan arvoksi asetetaan Luan antama päivitysfunktio. (LUA 2014 d.)

```
int LUA_SetUpdate(lua_State *S)
{
    //If the value at the given index is a full userdata, returns its block address.
    LuaObject *p = (LuaObject*)lua_touserdata(S, 1);

    //Pushes a light userdata onto the stack
    lua_pushlightuserdata(S, p);

    //Pushes onto the stack the value of LUA_REGISTRYINDEX[lightuserdata]
    lua_gettable(S, LUA_REGISTRYINDEX);
    lua_insert(S, -2);

    // Does the equivalent to table["update"]=function
    lua_setfield(S, -2, "update");

    return 0;
}
```

Kuva 8. LUA_SetUpdate-funktio moottorissa (Visual Studio 2010)

Luan päivitysfunktio asetetaan saatuun tauluun käyttämällä funktiota lua_setfield. Sen käytön vaatimuksena on, että tauluun asetettavan arvon on oltava Lua-pinossa päällimmäisenä. Päällimmäisenä osana on lua_gettablen jälkeen rekisteritaulun arvona oleva tyhjä taulu. Taulu siirretään alla olevan päivitysfunktion kohdalle, jolloin päivitysfunktio saadaan pinon päälle. (LUA 2014 d.)

Lua-komentosarjat

Lua-komentosarjoina kirjoitettiin käytännön osuudessa käytettävä objekti ja sen toiminnat. Liitteessä 1 nähdään, että Lua-tiedostoina on kirjoitettu main-tilan lisäksi luokat Ball ja GameArea. Luan näkökulmasta luokka Ball on taulu, joka pitää sisällään sijaintia ja suuntaa kuvaavia tauluja sekä niiden arvoja muuttavia funktioita. Taulu GameArea tarkastaa annetun taulun (Ball) sisältämän sijaintitaulun suhteen rajattuun alueeseensa. GameArean isintarkastusmetodia kutsutaan Ball-luokan metodissa update. (Ball-luokan update-metodia ei tule sekoittaa moottorin puoleisen rekisteritaulun arvossa olevaan ”update”-merkkijonoavaimeen.)

Komentosarjan main-tilassa käytetään moottorin puolella rekisteröityjä funktioita. Tauluun *a* tallennetaan NewLuaObject-funktion palauttaman olion viittaus. Samaa taulua *a* käytetään funktion SetUpdate parametrinä. Toiseen parametriin on kirjoitettu funktio, joka asetetaan rekisteritaulun arvona olevan taulun ”update”-avaimen arvoksi. Funktio kertoo moottorille Ball-luokan update-metodin jälkeisistä muutoksista käyttämällä rekisteröityä funktiota MoveObject.

6.6 Testaus

Opinnäytetyön alussa pohdittiin, kuinka pitkälle Lua-kielen käytössä voisi päästä annetun opinnäytetyön kirjoittamisen ajan aikana. Käytännön osuudessa selvisi, että testiprojektissa onnistuttiin luomaan moottorin 3D-objekti sekä hallitsemaan sitä Lua-komentosarjalla. Kirjoitusprosessin aikana ilmeni kuitenkin joitakin ongelmia. Testatessa liikkeen päivitystä selvisi, että valitsemani Lua-binääri ei tukenut Luan require-komentoa. Näin ollen jouduin siirtämään erillisissä lua-tiedostoissa olevat luokkani samaan tiedostoon. Näin syntyi *main2.lua*-tiedosto, jota kutsutaan moottorin main osiossa.

Koodin testauksessa ilmenevistä virheilmoituksista melkein kaikki olivat Lua-komentosarjoissa. Nämä olivat enimmäkseen syntaksivirheitä, jotka syntyivät siirtyessäni C++-koodin kirjoittamisesta Lua-komentosarjan kirjoittamiseen. Virheet olivat helppo löytää ja korjata, kiitos Luan tarkan virheidenjäljityksen. Toinen useita virheitä tuottavia käytännön osuuden kohtia oli Lua-pinon hallinta C++-koodin puolella.

6.7 Lopputulos

Testiprojektin lopputulos on moitteeton. Lua-kielen lisäys dynaamisesti linkitettynä kirjastona oli helppoa. Myös komentosarjoilla toteutettu moottorin toimintojen käyttö onnistui odotetusti. Objektin luominen, siirtäminen sekä päivitettävä liike saatiin lisättyä testiprojektiin. Kaikkia työlle annettuja tavoitteita ei kuitenkaan saavutettu. Testiprojektissa ehdittiin toteuttaa luodun objektin hallinta vain yhdelle objektille. Tavoitteisiin listattu useiden pallojen yhtäaikainen liikkeen päivitys ja toisiinsa reagointi jäi toteutumatta.

Tavoitteet itsessään olivat yksinkertaisia ja selkeitä. En kuitenkaan ottanut huomioon niiden yhdistämisen laajuutta. Päätettäessä testiprojektin tavoitteita minun olisi kannattanut luoda erillinen tehtävä objektien tunnistamiselle. Opettelin, miten moottori voi päästä käsiksi komentosarjojen tauluihin, mutta en selvittänyt, miten Lua-komentosarja voi saada tehokkaasti moottorin listauksesta tietyn objektin. Vaikka testiprojekti ei kelpaa sellaisenaan pelinkehitykseen, se toimii hyvin demona sille, mitä komentosarjoilla voi tehdä. Testiprojektiin kirjoitetut Lua-komentosarjat onnistuivat erinomaisesti. Lua-kielellä kirjoitettu yhteentörmäys ja suunnanvaihto toimivat myös hyvin.

7 POHDINTA

Komentosarjakielen toteuttaminen C++-pelimoottoriin on osoittautunut hyväksi valinnaksi opinnäytetyöni aiheeksi. Opinnäytetyön alussa toivoin voivani laajentaa Lua-kielen osaamistani. Nyt huomaan, että en ole oppinut vain Lua-kielestä vaan myös C++-kielestä sekä molempien kielien asemista pelimoottoreissa ja pelinkehityksessä. Opinnäytetyön käytännön osiossa toteutettu testiprojekti oli suurena osana oppimiskokemusta. Se on avannut soveltamismahdollisuuksia myöhempiä projekteja varten.

Vaikka lopputulos tuntuu pieneltä ja yksinkertaiselta, työn toteuttaminen ei ollut helppoa. Opinnäytetyön teoriaosuuden valmisteleminen ja kirjoittaminen olivat haasteellisimpia työprosessin vaiheita. Ne veivät myös suurimman osan opinnäytetyön tekemiseen tarkoitetusta ajasta. Jouduin lukemaan ja sisäistämään paljon materiaalia asioista, joista minulla oli joko vähän tai ei ollenkaan kokemusta. Tämä ongelma oli odotettavissa opinnäytetyön alkuvaiheilla, jonka takia pyrin varaamaan mahdollisimman paljon aikaa teorian toteuttamiseen. Varattu aika ei kuitenkaan ollut tarpeeksi, mistä johtuen jouduin karsimaan joitakin ominaisuuksia käytännön osuudesta.

Opinnäytetyön käytännön osuudessa ymmärsin, miksi niin monet Lua-kieleen liittyvät kolmannen osapuolen laajennukset keskittyvät helpottamaan C- ja Lua-objektien sitomista. Käytännön osuudessa olisi kannattanut jo alussa pysähtyä suunnittelemaan sitä, miten usean LuaObject-olioiden hallinta tapahtuisi moottorin näkökulmasta. Keskityin tekemisessäni vain siihen, mitä osaamiseni kulloinkin salli minun tekevän. Lua-komentosarjojen kirjoittaminen tuntui käytännön osuuden helpoimmalta osalta. Komentosarjakielen joustavuus teki pelikoodin kirjoittamisesta nopeaa. Se myös yksinkertaisti koodin kokonaisuutta. Koska Lua-kieli mahdollistaa usean muuttujan palauttaminen, pystyin jättämään turhien metodien luomisen pois koodista. Vastapainoisesti Lua-kielen sitominen C++-koodiin tuntui käytännön työn haasteellisimmalta osalta. En ollut aina tietoinen siitä, missä järjestyksessä Lua-pinon elementit olivat ja missä järjestyksessä tarvitsin niiden olevan toteuttaakseni haluamani funktiokutsut.

Opin paljon projektista, mutta suurin osa koodin todellisesta ymmärtämisestä tapahtui vasta kirjoittaessani sen toimintaa auki opinnäytetyöhön. Tulevaisuuden hyödyn kannalta testiprojektin tekeminen ja sen lopputulos ovat antaneet minulle tarpeellisen perusosaamisen ko-

mentosarjakielten ja pelimoottoreiden välisestä kommunikoinnista. Olen saanut myös tarpeellista osaamista itse Lua-komentosarjakielen kirjoittamisesta ja hallitsemisesta C++-ympäristössä.

LÄHTEET

Chaudhary, H. H. Thinking in Java. (Edition 2014) Complete Java Guide with Advance Java and Live Development. Noudettu, 2014, sivulta http://books.google.fi/books?id=61u6AwAAQBAJ&pg=PA517&dq=thinking+in+java&hl=fi&sa=X&ei=ES7aU4XFLqHnywOrnoHgCA&sqi=2&ved=0CCUQ6AEwAA#v=onepage&q=game&f=false

Code.Google. 2014. OOLua.
Noudettu, 2014, sivulta https://code.google.com/p/oolua/

Developer Network 2014 a. Walkthrough: Creating and Using a Static Library (C++)
Noudettu, 2014, sivulta http://msdn.microsoft.com/en-us/library/ms235627(v=vs.100).aspx

Developer Network 2014 b. DLLs in Visual C++.
Noudettu, 2014, sivulta http://msdn.microsoft.com/en-us/library/1ez7dh12.aspx

Dunbar, G. 2013. The Lua Tutorial.
Noudettu, 2014, sivulta http://www.gamedev.net/page/resources/_/technical/game-programming/the-lua-tutorial-r2999

Ford, J. L. Jr. 2007. Ruby Programming for the Absolute Beginner.
Noudettu, 2014, sivulta http://books.google.fi/books?id=zX0LAAAAQBAJ&pg=PT26&dq=scripting+language+pearl+ruby&hl=fi&sa=X&ei=xoLXU4nsEKv-ygO5tIHABQ&redir_esc=y#v=onepage&q=scripting&f=false

GAMEDEV. 2011. What are the pros and cons of incorporating Lua into a C++ game?.
Noudettu, 2014, sivulta http://gamedev.stackexchange.com/questions/18285/what-are-the-pros-and-cons-of-incorporating-lua-into-a-c-game

GDMag Staff. 2012. Front Line Award Winners Announced.
Noudettu, 2014, sivulta https://web.archive.org/web/20130615013638/http://www.gdmag.com/blog/2012/01/front-line-award-winners.php

Gregory, J. 2009. *Game Engine Architecture.*

Gregory, J. 2009. Game Engine Architecture.
Noudettu, 2014, sivulta http://forum.dronprogs.org/files_for_my_posts/books/GEA.pdf

HAVOK. 2014. Havok Physics.
Noudettu, 2014, sivulta http://www.havok.com/products/physics#sthash.cbIzZuay.dpuf

HELLOWORLDOPEN. 2014. Java is back? - programming language trends in hwo.
Noudettu, 2014, sivulta http://blog.helloworldopen.com/post/83402776977/java-is-back-programming-language-trends-in-hwo

Ierusalimschy, R., Figueiredo, L. H. & Celes, W. 2006.
Game Programming Gems 6 : 4.2 Binding C/C++ Objects to Lua.

Ierusalimschy, R., Figueiredo, L. H. & Celes, W. 2007.
The Evolution of Lua HOPL III

Ierusalimschy, R., Figueiredo, L. H. & Celes, W. The evolution of Lua.
Noudettu, 2014, sivulta http://www.lua.org/doc/hopl.pdf

JAVA. 2014. BeginnersGuideOverview.
Noudettu, 2014, sivulta https://wiki.python.org/moin/BeginnersGuide/Overview

Johnson, M. 2013. Including Lua 5.x in Visual Studio 2012 for Embedding in C++
Noudettu, 2014, sivulta http://codethinktank.blogspot.fi/2013/04/including-lua-5x-in-visual-studio-2012.html

LUA 2014 a. About.
Noudettu, 2014, sivulta http://www.lua.org/about.html

LUA 2014 b. Start.
Noudettu, 2014, sivulta http://www.lua.org/start.html

LUA 2014 c. Read me.
Noudettu, 2014, sivulta http://www.lua.org/manual/5.2/readme.html

LUA 2014 d. Manual.
Noudettu, 2014, sivulta http://www.lua.org/manual/5.2/manual.html

LuaBinaries 2014 a. Manual.
Noudettu, 2014, sivulta http://luabinaries.sourceforge.net/manual.html

LuaBinaries 2014 b. Overview.
Noudettu, 2014, http://luabinaries.sourceforge.net/index.html#over

LuaForWindows. 2013. Lua for Windows.
Noudettu, 2014, sivulta https://code.google.com/p/luaforwindows/

LuaJit. 2014. Overview.
Noudettu, 2014, sivulta http://luajit.org/luajit.html

LUAPLUS. 2014. What is LuaPlus?.
Noudettu, 2014, sivulta http://luaplus.org/

Petri. (2012). Making of Grimrock: Rapid Programming.
Noudettu, 2014, sivulta http://www.grimrock.net/2012/07/25/making-of-grimrock-rapid-programming/

Rabin, S. (2010). Introduction to Game Development.
Noudettu, 2014, sivulta
http://books.google.fi/books?id=bb8LAAAAQBAJ&pg=PA203&dq=game+scripting+language+Python&hl=fi&sa=X&ei=gpPXU72uGoroywOHioCYAw&redir_esc=y#v=onepage&q=game%20scripting%20language%20Python&f=false

RASTERBAR. (2014). Luabind.
Noudettu, 2014, sivulta http://www.rasterbar.com/products/luabind.html

Rossum, G. (2014). Guido van Rossum – Personal Home Page.
Noudettu, 2014, sivulta: https://www.python.org/~guido/

Ruby 2014 a. Ruby 2.1.2 is released.
Noudettu, 2014, sivulta https://www.ruby-lang.org/en/news/2014/05/09/ruby-2-1-2-is-released/

Ruby 2014 b. About Ruby.
Noudettu, 2014, sivulta https://www.ruby-lang.org/en/about/

Scott, M. L. (2009). Programming Language Pragmatics.
Noudettu, 2014, sivulta
http://books.google.fi/books?id=GBISkhhrHh8C&pg=PA650&dq=scripting+language&hl=fi&sa=X&ei=gnHXU_6lNaHMygOLooGQDA&redir_esc=y#v=onepage&q=scripting%20language&f=false

TECHNEILLOGY. (2012). Compiling Lua with Visual Studio 2010.
Noudettu, 2014, sivulta http://techneilogy.blogspot.fi/2012/02/compiling-lua-with-visual-studio-2010.html

Varanese, A. (2003) Game Scripting Mastery.

Vogel, L. (2014). Introduction to Java programming - Tutorial.
Noudettu, 2014, sivulta
http://www.vogella.com/tutorials/JavaIntroduction/article.html#javaintroduction

Ward, J. (2008). What is a Game Engine?.
Noudettu, 2014, sivulta:
http://www.gamecareerguide.com/features/529/what_is_a_game.php

WeboPedia. (2014). Scripting language.
Noudettu, 2014, sivulta:
http://www.webopedia.com/TERM/S/scripting_language.html

LIITTEET

Liite 1. Komentosarjan *main2.lua* koodi.

```
---[[ AreaScrtipt.lua – Start
-- Script to define game area
GameArea = {
                        minX = -37,
                        minY = -6,
                        maxX = 37,
                        maxY = 96
                }

-- Checks if position of given object is out of game area
function GameArea:isIn(object)

        --Check if parameter has value
        if object.position == nil then
                print("\tERROR in isIn! Taken param is nil!")
                return
        end

        -- Returns result of collision and new direction for an object
        local pos = object.position
        local vel = object.velocity
        local result = false

        if (pos.x > self.maxX and vel.x > 0) or (pos.x < self.minX and vel.x < 0) then
                result, vel = true, {vel.x*(-1), vel.y};
        elseif (pos.y > self.maxY and vel.y > 0) or (pos.y < self.minY and vel.y < 0) then
                result, vel = true, {vel.x, vel.y*(-1)};
        end

        return result, vel;
end
--]]AreaScrtipt.lua - End
```

```lua
---[[ BallScrtipt.lua - Start
Ball = {
                        velocity = {x = 1, y = 1},
                        position = {x = 0, y = 0, z = 0}
        }

-- Creates new instance (From Programming in Lua (2013), chapter 16.1 Classes)
function Ball:new(o)
        o = o or {} -- create table if user does not provide one
        setmetatable(o, self)
        self.__index = self
        return o
end

-- Sets new position
function Ball:updatePos()
        local p = self.position;
        local v = self.velocity;

        self.position.x = p.x + v.x;
        self.position.y = p.y + v.y;
end

-- Sets new velocity from given table
function Ball:setVel(direction)

        if direction == nil then
                print("\tERROR in setVel! Taken param is nil!")
        end

        self.velocity.x = direction[1];
        self.velocity.y = direction[2];
        self:updatePos()
        -- empty table
        direction = nil
end

function Ball:update()
        -- check collision
        local result, dir = GameArea:isIn(self)

        -- if collisions happened, change the direction of velocity
        if result == true then
                print("update: OUT OF AREA! Change of direction!")
                self:setVel(dir)
        end

        -- move object
        self:updatePos()
end
--]]BallScrtipt.lua - End
```

```
-- MAIN - START
B = Ball:new()
a = NewLuaObject("ball.obj", "texture.png")

SetUpdate(a, function()
        B:update()
        MoveObject(B.position.x, B.position.y)
end)
```

Liite 2. Moottorin *TestCase.cpp* koodi.

```
int LUA_NewLuaObject(lua_State *S) {
	//Checks and takes parameters from Lua stack
	luaL_checktype(S, -1, LUA_TSTRING);
	const std::string textureFilename = lua_tostring(S, -1);
	luaL_checktype(S, -2, LUA_TSTRING);
	const std::string vertexFilename = lua_tostring(S, -2);

	//A new block of memory is allocated with the given size,
	//a new full userdata with the block address is pushed onto the stack
	LuaObject *newObject = (LuaObject*)lua_newuserdata(S, sizeof(LuaObject));
	new (newObject)LuaObject(vertexFilename, true, textureFilename, S);

	lua_pushlightuserdata(S, newObject);
	lua_newtable(S);
	lua_settable(S, LUA_REGISTRYINDEX);

	//Adds object to GameObject vector
	gamePointerObjects.push_back((GameObject*)newObject);

	return 1;
}

int LUA_MoveObject(lua_State *S) {
	luaL_checktype(S, -1, LUA_TNUMBER);
	const float yPos = lua_tointeger(S, -1);
	luaL_checktype(S, -2, LUA_TNUMBER);
	const float xPos = lua_tointeger(S, -2);

	//Set new position to object in list
	gamePointerObjects.at(0)->SetPosition(xPos, yPos, 0.0f);

	return 0;
}

int LUA_SetUpdate(lua_State *S) {
	//If full userdata, returns its block address.
	LuaObject *p = (LuaObject*)lua_touserdata(S, 1);

	//Pushes a light userdata onto the stack
	lua_pushlightuserdata(S, p);

	//Pushes onto the stack the value of LUA_REGISTRYINDEX[lightuserdata]
	lua_gettable(S, LUA_REGISTRYINDEX);
	lua_insert(S, -2);

	// Does the equivalent to table["update"]=function
	lua_setfield(S, -2, "update");

	return 0;
}

void init(lua_State *S) {
	lua_register(S, "NewLuaObject", LUA_NewLuaObject);
	lua_register(S, "MoveObject", LUA_MoveObject);
	lua_register(S, "SetUpdate", LUA_SetUpdate);
}
```

Liite 3. Moottorin *main.cpp* koodi.

```
void init(lua_State *S);

/**
* Initialize all objects of game here.
**/
void GameMain()
{
	// Alusta kamera
	gameCamera = new GameCamera();
	gameCamera->SetPosition(0, -45, -100);

//Lua --- start

	// Asennus
	lua_State *S = luaL_newstate();
	luaL_openlibs(S);

	// Rekisteröinti
	init(S);

	// Suoritus
	if(luaL_loadfile(S, "GameScripts/main2.lua") || lua_pcall(S,0,0,0))
	{
		luaL_checktype(S, -1, LUA_TSTRING);
		std::string error = lua_tostring(S, -1);
		std::cout << "ERROR: " << error << std::endl;
	}

 //Lua --- end

}

/**
* All methods that need updating are put here.
**/
void UpdateLoop()
{
	for(int i = 0; i < LuaObject::objectPtrs.size(); i++)
	{
		LuaObject::objectPtrs.at(i)->Update();
	}
}
```